

Уильям Стивенс

UNIX: взаимодействие процессов

МАСТЕР-КЛАСС

Всему сообществу Usenet, где можно найти много ЧаВо с ответами на все вопросы и случаи жизни

Предисловие

Большинство нетривиальных программ пишутся с использованием одной из форм межпроцессного взаимодействия (IPC — Interprocess Communication). Это естественное следствие принципа разработки, заключающегося в том, что лучше создавать приложение, состоящее из отдельных взаимодействующих элементов, чем одну большую программу. Исторически разработка приложений пережила следующие этапы развития:

1. Сначала были большие цельные программы, которые делали все необходимое. Отдельные части программы реализовывались в виде функций, обменивавшихся информацией через аргументы, возвращаемые значения и глобальные переменные.
2. Потом стали создаваться небольшие программы, взаимодействующие друг с другом посредством различных средств IPC. Многие стандартные утилиты Unix были разработаны именно таким образом, причем для передачи информации использовались каналы интерпретатора.
3. Наконец, сейчас появилась возможность писать цельные программы, состоящие из взаимодействующих между собой потоков. В данном случае мы все равно применяем термин IPC, хотя процесс имеется только один.

Комбинация последних двух вариантов также возможна: несколько процессов, каждый из которых состоит из нескольких потоков, вполне могут взаимодействовать между собой.

Мы описываем возможность разделения решаемых задач между несколькими процессами или даже между потоками одного процесса. В мультипроцессорной системе такое приложение сможет выполняться гораздо быстрее, поэтому разделение задач между процессами способно повысить его быстродействие.

В этой книге подробно описываются четыре формы IPC:

1. Передача сообщений (каналы, FIFO, очереди сообщений).
2. Синхронизация (взаимные исключения, условные переменные, блокировки чтения-записи, блокировка файлов и записей, семафоры).
3. Разделяемая память (неименованная и именованная).
4. Удаленный вызов процедур (двери, Sun RPC).

Здесь не рассматриваются вопросы написания программ, взаимодействующих по сети. Такая форма взаимодействия обычно подразумевает использование интерфейса сокетов и стека протоколов TCP/IP; эти темы были подробно разобраны в первом томе книги ([24]).

Нам могут возразить, что средства IPC, не предназначенные для взаимодействия по сети, вообще не следует использовать и что вместо этого следует изначально разрабатывать приложения с расчетом на использование в сети. Однако на практике средства IPC, работающие только в пределах одного узла, функционируют гораздо быстрее, чем сетевые, да и программы с их использованием оказываются проще. Разделяемая память и средства синхронизации обычно не могут использоваться по сети — они доступны только в пределах одного узла. Опыт и история показывают, что существует потребность в наличии как несетевых, так и сетевых форм IPC.

В этой книге используется материал первого тома и других моих книг:

- UNIX Network Programming, том 1, 1998 [24];
- Advanced Programming in the UNIX Environment, 1992 [21];
- TCP/IP Illustrated, том 1, 1994 [22];
- TCP/IP Illustrated, том 2, написанной в соавторстве с Гари Райтом (Gary Wright), 1995, [27];
- TCP/IP Illustrated, том 3, 1996 [23].

Может показаться странным, что я описываю средства IPC в книге, заглавие которой содержит слова «Network Programming». Замечу, что IPC часто используется и в сетевых приложениях. Как говорилось в предисловии к книге «UNIX Network Programming» 1990 года издания, «для понимания методов разработки сетевых приложений необходимо понимание средств межпроцессного взаимодействия (IPC)».

Этот том содержит полностью переписанные главы 3 и 18 книги «UNIX Network Programming» 1990 года издания. Если подсчитать количество слов, объем материала увеличился в пять раз. Ниже

перечислены основные отличия данного издания:

- В дополнение к трем формам System V IPC (очереди сообщений, семафоры, разделяемая память) рассматриваются более новые функции Posix, реализующие эти же три формы IPC. О стандартах Posix более подробно говорится в разделе 1.7. В будущем можно ожидать перехода к использованию функций Posix, обладающих определенными преимуществами по сравнению с аналогами System V.
- Рассматриваются средства синхронизации Posix: взаимные исключения, условные переменные, блокировки чтения-записи. Эти средства могут использоваться для синхронизации потоков или процессов и часто привлекаются для обеспечения синхронизации доступа к разделяемой памяти.
- В этом томе предполагается наличие поддержки потоков Posix (Pthreads), и многие примеры написаны с использованием многопоточного (а не многопроцессного) программирования.
- Описание именованных и неименованных каналов и блокировок записей основано на их определениях в стандарте Posix.
- В дополнение к описанию средств IPC и примерам их использования я также привожу примеры реализации очередей сообщений, блокировок чтения-записи и семафоров Posix (все это может быть скомпилировано в пользовательские библиотеки). Эти реализации задействуют множество разных средств одновременно. Например, одна из реализаций семафоров Posix использует взаимные исключения, условные переменные и отображение в память. В комментариях отмечаются важные моменты, которые следует учитывать при разработке приложений (ситуации гонок, обработка ошибок, утечка памяти, использование списков аргументов переменной длины). Понимание реализации какого-либо средства ведет к лучшему его использованию.
- При описании RPC основное внимание уделяется пакету Sun RPC. Рассказ предваряется описанием нового интерфейса дверей в Solaris, который похож на RPC, но используется только в пределах одного узла. Описание дверей является как бы введением, в котором описываются важные вопросы вызова процедур в других процессах без необходимости учитывать особенности сетевой реализации.

Эта книга может использоваться как учебник по IPC или как справочник для опытных программистов. Текст разделен на четыре части:

- передача сообщений;
- синхронизация;
- разделяемая память;
- удаленный вызов процедур.

Возможно, некоторые читатели будут интересоваться содержимым конкретных подразделов. Большая часть глав может читаться совершенно независимо от остальных, хотя в главе 2 объединены общие особенности средств Posix IPC, в главе 3 — System V IPC, а глава 12 является введением в разделяемую память (как Posix, так и System V). Всем читателям настоятельно рекомендуется прочесть главу 1, в особенности раздел 1.6, в котором описываются используемые в книге функции-обертки. Главы, описывающие средства Posix IPC, могут читаться отдельно от глав, посвященных System V IPC. Описание каналов и блокировок записей стоит особняком. Две главы, посвященные удаленному вызову процедур, также могут читаться отдельно от прочих.

Подробный индекс упрощает использование книги в качестве справочника. Для читающих текст в случайном порядке приводятся многочисленные перекрестные ссылки на сходный материал.

Исходный код всех примеров можно загрузить с домашней страницы автора (адрес — в конце предисловия). Лучший способ изучить IPC — это изменить программы из примеров или даже улучшить их. Написание программ лучше всего способствует усвоению концепций и методов. В конце каждой главы даются упражнения, решения к большей части которых даны в приложении Г.

Список замеченных опечаток можно также найти на домашней странице автора.

Хотя на обложке книги стоит имя только одного автора, в ее создании участвовало множество людей. Прежде всего это члены семьи автора, которые смирились с ушедшими на ее написание часами. Еще раз спасибо, Салли, Билл, Эллен и Дэвид.

Спасибо всем, кто помогал работать с содержимым книги. Ваша помощь была просто неоценимой (135 печатных страниц). Вы исправляли ошибки, отмечали недостаточную четкость пояснений, предлагали другие объяснения и варианты программ. Спасибо вам, Гевин Боуи, Аллен Бриггс, Дейв Бутенхов, Ван-Тех Чанг, Крис Клилэнд, Боб Фриснан, Эндрю Гиерт, Скотт Джонсон, Марти Леиснер,

Ларри Мак-Вой, Крейг Метз, Боб Нельсон, Стив Рэго, Джим Рейд, Свами К. Ситарама, Джон К. Снейдер, Иан Ланс Тейлор, Рик Тир и Энди Такер.

Мне помогали и те, кто отвечал на мои электронные письма, в которых порой было множество вопросов. Ваши ответы помогли сделать книгу более точной и ясной: Дэвид Баусум, Дейв Бутенхов, Билл Голмейстер, Макеш Кэкер, Брайан Керниган, Ларри МакВой, Стив Рэго, Кейт Скорвран, Барт Смаалдерс, Энди Такер и Джон Уэйт.

Отдельная благодарность Ларри Рафски из GSquared. Спасибо, как обычно, говорю я NOAO, Сиднею Вульфу, Ричарду Вульфу и Стиву Гранди за возможность работать с их сетями и компьютерами. Джим Баунд, Мэтт Томас, Мэри Клаутер и Барб Glover из Digital Equipment Corp. предоставили систему Alpha, на которой выполнялась большая часть примеров данной книги. Часть программ была протестирована и в других системах. Спасибо Майклу Джонсону из Red Hat Software (за новейшие версии Red Hat Linux), Дейву Маркуардту и Джесси Хауг за компьютер RS/6000 и доступ к последним версиям AIX.

Благодарю сотрудников Prentice Hall — редактора Мэри Франц вместе с Норин Регина, Софи Папаниколау и Патти Гуэрриери — за помощь, в особенности в соблюдении сроков.

Оригинал-макет этой книги был подготовлен на языке PostScript. Форматирование осуществлялось с помощью замечательного пакета groff (автор — Джеймс Кларк) на SparcStation под управлением Solaris 2.6. (Сведения о смерти groff сильно преувеличены). Я набил все 138 897 слов книги в редакторе vi, создал 72 рисунка с помощью программы grc (используя макросы Гари Райта), сделал 35 таблиц с помощью программы gtbl, подготовил индекс (с помощью сценариев на языке awk, написанных Джоном Бентли и Брайаном Керниганом) и сверстал все это вместе. Программа Дейва Хэнсона loom, пакет GNU indent и сценарии Гари Райта помогли добавить в книгу 8046 строк исходного кода на языке C.

С нетерпением жду комментариев, предложений и сообщений о замеченных опечатках.

W. Richard Stevens Tucson, Arizona July 1998

rsteven@kohala.com <http://www.kohala.com/~rsteven>

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу
<http://www.piter.com/download>.

На web-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

1.1. Введение

Аббревиатура IPC расшифровывается как *interprocess communication*, то есть взаимодействие процессов. Обычно под этим понимается передача сообщений различных видов между процессами в какой-либо операционной системе. При этом могут использоваться различные формы синхронизации, требуемой современными видами взаимодействия, осуществлямыми, например, через разделяемую память.

В процессе развития операционных систем семейства Unix за последние 30 лет методы передачи сообщений эволюционировали следующим образом:

- Каналы (pipes — глава 4) были первой широко используемой формой взаимодействия процессов, доступной программам и пользователю (из интерпретатора команд). Основным недостатком каналов является невозможность их использования между процессами, не имеющими общего родителя (ancestor), но этот недостаток был устранен с появлением именованных каналов (named pipes), или каналов FIFO (глава 4).
- Очереди сообщений стандарта System V (System V message queues — глава 4) были добавлены к ядрам System V в начале 80-х. Они могут использоваться для передачи сообщений между процессами на одном узле вне зависимости от того, являются ли эти процессы родственными. Несмотря на сохранившийся префикс «System V», большинство современных версий Unix, включая и те, которые не произошли от System V, поддерживают эти очереди.

ПРИМЕЧАНИЕ

В отношении процессов Unix термин «родство» означает, что у процессов имеется общий предок. Подразумевается, что процессы, являющиеся родственниками, были созданы этим процессом-предком с помощью одной или нескольких «вилок» (forks). Простейшим примером будет вызов fork некоторым процессом дважды, что приведет к созданию двух порожденных процессов. Тогда можно говорить о родстве этих процессов между собой. Естественно, каждый порожденный процесс является родственником породившего. Родитель может позаботиться о возможности взаимодействия с порожденным процессом (создав канал или очередь сообщений) перед вызовом fork, и этот объект IPC будет унаследован порожденным процессом. Более подробно о наследовании объектов IPC рассказано в табл. 1.4. Нужно также отметить, что все процессы Unix теоретически являются потомками процесса init, который запускает все необходимое в процессе загрузки системы (bootstrapping). С практической точки зрения отсчет родства процессов лучше вести с оболочки (login shell) и всех процессов, ею созданных. В главе 9 [24] рассказано о сеансах и родственных отношениях процессов более подробно.

ПРИМЕЧАНИЕ

Примечания вроде этого будут использоваться нами для того, чтобы уточнять особенности реализации, давать исторические справки и полезные советы.

- Очереди сообщений Posix (Posix message queues — глава 5) были добавлены в стандарт Posix (1003.1b-1993, о котором более подробно рассказано в разделе 1.7). Они могут использоваться для взаимодействия родственных и неродственных процессов на каком-либо узле.
- Удаленный вызов процедур (remote procedure calls — RPC, часть 5) появился в 80-х в качестве средства для вызова функций на одной системе (сервере) программой, выполняемой на другой системе (клиенте). Это средство было разработано в качестве альтернативы для упрощения сетевого программирования. Поскольку между клиентом и сервером обычно передается информация (передаются аргументы для вызова функции и возвращаемые значения) и поскольку удаленный вызов процедур может использоваться между клиентом и сервером на одном узле, RPC можно также считать одной из форм передачи сообщений.

Интересно также взглянуть на эволюцию различных форм синхронизации в процессе развития Unix:

- Самые первые программы, которым требовалась синхронизация (чаще всего для предотвращения одновременного изменения содержимого файла несколькими процессами), использовали особенности файловой системы, некоторые из которых описаны в разделе 9.8,
- Возможность блокирования записей (record locking — глава 9) была добавлена к ядрам Unix в начале 80-х и стандартизована в версии Posix.1 в 1988.
- Семафоры System V (System V semaphores — глава 11) были добавлены вместе с возможностью совместного использования памяти (System V shared memory — глава 14) и одновременно с очередями сообщений System V (начало 80-х). Эти IPC поддерживаются большинством современных версий Unix.

- Семафоры Posix (Posix semaphores — глава 10) и разделяемая память Posix (Posix shared memory — глава 13) были также добавлены в стандарт Posix (1003.1b-1993, который ранее упоминался в связи с очередями сообщений Posix).
- Взаимные исключения и условные переменные (mutex, conditional variable — глава 7) представляют собой две формы синхронизации, определенные стандартом программных потоков Posix (Posix threads, Pthreads — 1003.1c-1995). Хотя обычно они используются для синхронизации между потоками, их можно применять и при организации взаимодействия процессов.
- Блокировки чтения-записи (read-write locks — глава 8) представляют собой дополнительную форму синхронизации. Она еще не включена в стандарт Posix, но, вероятно, скоро будет.

В традиционной модели программирования Unix в системе могут одновременно выполняться несколько процессов, каждому из которых выделяется собственное адресное пространство. Это иллюстрирует рис. 1.1.



Рис. 1.1. Совместное использование информации процессами

1. Два процесса в левой части совместно используют информацию, хранящуюся в одном из объектов файловой системы. Для доступа к этим данным каждый процесс должен обратиться к ядру (используя функции read, write, lseek, write, lseek и аналогичные). Некоторая форма синхронизации требуется при изменении файла, для исключения помех при одновременной записи в файл несколькими процессами и для защиты процессов, читающих из файла, от тех, которые пишут в него.

2. Два процесса в середине рисунка совместно используют информацию, хранящуюся в ядре. Примерами в данном случае являются канал, очередь сообщений или семафор System V. Для доступа к совместно используемой информации в этом случае будут использоваться системные вызовы.

3. Два процесса в правой части используют общую область памяти, к которой может обращаться каждый из процессов. После того как будет получен доступ к этой области памяти, процессы смогут обращаться к данным вообще без помощи ядра. В этом случае, как и в первом, процессам, использующим общую память, также требуется синхронизация.

Обратите внимание, что ни в одном из этих случаев количество взаимодействующих процессов не ограничивается двумя. Любой из описанных методов работает для произвольного числа взаимодействующих процессов. На рисунке мы изображаем только два для простоты.

Потоки

Хотя концепция процессов в системах Unix используется уже очень давно, возможность использовать несколько потоков внутри одного процесса появилась относительно недавно. Стандарт потоков Posix.1, называемый Pthreads, был принят в 1995 году. С точки зрения взаимодействия процессов все потоки одного процесса имеют общие глобальные переменные (то есть поточной модели свойственно использование общей памяти). Однако потокам требуется синхронизация доступа к глобальным данным. Вообще, синхронизация, не являясь собственно формой IPC, часто используется совместно с различными формами IPC для управления доступом к данным.

В этой книге описано взаимодействие между процессами и между потоками. Мы предполагаем наличие среды, в которой поддерживается многопоточное программирование, и будем использовать выражения вида «если канал пуст, вызывающий поток блокируется до тех пор, пока какой-нибудь другой поток не произведет запись в канал». Если система не поддерживает потоки, можно в этом предложении заменить «потоки» на «процессы» и получится классическое определение блокировки в Unix, возникающей при считывании из пустого канала командой `read`. Однако в системе, поддерживающей потоки, блокируется только поток, запросивший данные из пустого канала, а все остальные потоки процесса будут продолжать выполняться. Записать данные в канал сможет другой поток этого же процесса или какой-либо поток другого процесса.

В приложении Б сведены некоторые основные характеристики потоков и дано описание пяти основных функций Pthread, используемых в программах этой книги.

1.3. Живучесть объектов IPC

Можно определить живучесть (persistence) любого объекта IPC как продолжительность его существования. На рис. 1.2 изображены три возможные группы, к которым могут быть отнесены объекты по живучести.



Рис. 1.2. Живучесть объектов IPC

1. Объект IPC, живучесть которого определяется процессом (process-persistent), существует до тех пор, пока не будет закрыт последним процессом, в котором он еще открыт. Примером являются неименованные и именованные каналы (pipes, FIFO).

2. Объект IPC, живучесть которого определяется ядром (kernel-persistent), существует до перезагрузки ядра или до явного удаления объекта. Примером являются очереди сообщений стандарта System V, семафоры и разделяемая память. Живучесть очередей сообщений Posix, семафоров и разделяемой памяти должна определяться по крайней мере ядром, но может определяться и файловой системой в зависимости от реализации.

3. Объект IPC, живучесть которого определяется файловой системой (filesystem-persistent), существует до тех пор, пока не будет удален явно. Его значение сохраняется даже при перезагрузке ядра. Очереди сообщений Posix, семафоры и память с общим доступом обладают этим свойством, если они реализованы через отображаемые файлы (так бывает не всегда).

Следует быть аккуратным при определении живучести объекта IPC, поскольку она не всегда очевидна. Например, данные в канале (pipe) обрабатываются ядром, но живучесть каналов определяется процессами, а не ядром, потому что после того, как последний процесс, которым канал был открыт на чтение, закроет его, ядро сбросит все данные и удалит канал. Аналогично, хотя каналы FIFO и обладают именами в файловой системе, живучесть их также определяется процессами, поскольку все данные в таком канале сбрасываются после того, как последний процесс, в котором он был открыт, закроет его.

В табл. 1.1 сведена информация о живучести перечисленных ранее объектов IPC.

Таблица 1.1. Живучесть различных типов объектов IPC

Обратите внимание, что ни один тип IPC в этой таблице не обладает живучестью, определяемой файловой системой. Мы уже упомянули о том, что три типа объектов IPC в стандарте Posix *могут* иметь этот тип живучести в зависимости от реализации. Очевидно, что запись данных в файл обеспечивает живучесть, определяемую файловой системой, но обычно IPC таким образом не реализуются. Большая часть объектов IPC не предназначена для того, чтобы существовать и после перезагрузки, потому что ее не переживают процессы. Требование живучести, определяемой файловой системой, скорее всего, снизит производительность данного типа IPC, а обычно одной из задач разработчика является именно обеспечение высокой производительности.

1.4. Пространства имен

Если два неродственных процесса используют какой-либо вид IPC для обмена информацией, объект IPC должен иметь имя или идентификатор, чтобы один из процессов (называемый обычно сервером — server) мог создать этот объект, а другой процесс (обычно один или несколько клиентов — client) мог обратиться к этому конкретному объекту.

Программные каналы (pipes) именами не обладают (и поэтому не могут использоваться для взаимодействия между неродственными процессами), но каналам FIFO сопоставляются имена в файловой системе, являющиеся их идентификаторами (поэтому каналы FIFO могут использоваться для взаимодействия неродственных процессов). Для других типов IPC, рассматриваемых в последующих главах, используются дополнительные соглашения об именовании (naming conventions). Множество возможных имен для определенного типа IPC называется его пространством имен (name space). Пространство имен — важный термин, поскольку для всех видов IPC, за исключением простых каналов, именем определяется способ связи клиента и сервера для обмена сообщениями.

В табл. 1.2 сведены соглашения об именовании для различных видов IPC.

Таблица 1.2. Пространства имен для различных типов IPC

Здесь также указано, какие формы IPC содержатся в стандарте Posix.1 1996 года и какие были включены в стандарт Unix 98. Об обоих этих стандартах более подробно рассказано в разделе 1.7. Для сравнения мы включили в эту таблицу три типа сокетов, которые подробно описаны в [24]. Обратите внимание, что интерфейс сокетов (Application Program Interface — API) стандартизируется рабочей группой Posix.1g и должен в будущем стать частью стандарта Posix.1.

Хотя стандарт Posix.1 и дает возможность использования семафоров, их поддержка не является обязательной для производителей. В табл. 1.3 сведены функции, описанные в стандартах Posix.1 и Unix 98. Каждая функция может быть обязательной (mandatory), неопределенной (not defined) или необязательной (дополнительной — optional). Для необязательных функций мы указываем имя константы (например, _POSIX_THREADS), которая будет определена (обычно в заголовочном файле <unistd.h>), если эта функция поддерживается. Обратите внимание, что Unix 98 содержит в себе Posix.1 в качестве подмножества.

Таблица 1.3. Доступность различных форм IPC

1.5. Действие команд fork, exec и _exit на объекты IPC

Нам нужно достичь понимания действия функций fork, exec и _exit на различные формы IPC, которые мы обсуждаем (последняя из перечисленных функций вызывается функцией exit). Информация по этому вопросу сведена в табл. 1.4.

Большинство функций описаны далее в тексте книги, но здесь нужно сделать несколько замечаний. Во-первых, вызов fork из многопоточного процесса (multithreaded process) приводит к беспорядку в безымянных переменных синхронизации (взаимных исключений, условных переменных, блокировках и семафорах, хранящихся в памяти). Раздел 6.1 книги [3] содержит необходимые детали. Мы просто отметим в добавление к таблице, что если эти переменные хранятся в памяти с общим доступом и создаются с атрибутом общего доступа для процессов, они будут доступны любому процессу, который может обращаться к этой области памяти. Во-вторых, три формы IPC System V не могут быть открыты или закрыты. Из листинга 6.6 и упражнений 11.1 и 14.1 видно, что все, что нужно знать, чтобы получить доступ к этим трем формам IPC, — это идентификатор. Поэтому они доступны всем процессам, которым известен этот идентификатор, хотя для семафоров и памяти с общим доступом требуется некая особая обработка.

Таблица 1.4. Действие fork, exec и _exit на IPC

В любой реальной программе при любом вызове требуется проверка возвращаемого значения на наличие ошибки. Поскольку обычно работа программ при возникновении ошибок завершается, мы можем сократить объем текста, определив функции-обертки (wrapper functions), которые осуществляют собственно вызов функции, проверяют возвращаемое значение и завершают работу при возникновении ошибок. В соответствии с соглашениями имени функций-оберток совпадают с именами самих функций, за исключением первой буквы, которая делается заглавной, например

Пример функции-обертки приведен в листинге 1.1

Если в тексте вы встретите имя функции, начинающееся с заглавной буквы, знайте: это наша собственная функция-обертка. Она вызывает функцию с тем же именем, начинающимся со строчной буквы. Функция-обертка приводит к завершению работы процесса с выводом сообщения об ошибке, если таковая возникает.

При описании исходного кода, включенного в книгу, мы всегда говорим о вызываемой функции самого низкого уровня (например, `sem_post`), а не о функции-обертке (например, `Sem_post`). Аналогично в алфавитном указателе приведены имена самих функций, а не оберток к ним.

ПРИМЕЧАНИЕ

Вышеприведенный формат исходного кода используется во всем тексте. Все непустые строки нумеруются. Текст, описывающий разделы кода, начинается с номеров первой и последней строк на пустом поле слева. Иногда перед абзацем текста присутствует краткий заголовок, набранный полужирным шрифтом, излагающий основное содержание описываемого кода.

В начале кода указывается имя исходного файла. В данном примере — это файл `wrapunix.c` в каталоге `lib`. Поскольку исходный код всех примеров этой книги распространяется свободно (см. предисловие), вы можете легко найти требуемый файл. Компиляция, выполнение и особенно изменение этих программ в процессе чтения книги — лучший способ изучить концепции взаимодействия процессов.

Хотя может показаться, что использовать такие функции-обертки не слишком выгодно, вы избавитесь от этого заблуждения в главе 7, где мы обнаружим, что функции для работы с потоками (thread functions) не присваивают значение стандартной переменной Unix `errno` при возникновении ошибки; вместо этого код ошибки просто возвращается функцией. Это означает, что при вызове функции `pthread` мы должны каждый раз выделять память под переменную, сохранять в ней возвращаемое функцией значение, а затем устанавливать значение переменной `errno` равным этой переменной, прежде чем вызывать функцию `err_sys` (листинг B.4). Чтобы не загромождать текст фигурными скобками, мы используем оператор языка Си «запятая» (`,`) и совмещаем присваивание значения переменной `errno` и вызов `err_sys` в одном операторе, как в нижеследующем примере:

Альтернативой является определение новой функции обработки ошибок, принимающей код ошибки в качестве аргумента. Однако мы можем сделать этот фрагмент кода гораздо более читаемым, записав

где используется наша собственная функция-обертка, приведенная в листинге 1.2.

ПРИМЕЧАНИЕ

Аккуратно используя возможности языка Си, мы могли бы применять макросы вместо функций, что увеличило бы скорость выполнения программ, но эти функции-обертки редко бывают (если вообще бывают) узким местом.

Наше соглашение о замене первой буквы имени функции на заглавную является компромиссом. Рассматривалось много других форм записи: использование префикса `e` ([10, с. 182]), суффикса `_e` и т. д. Наш вариант кажется наименее отвлекающим внимание и одновременно дающим визуальное указание на то, что вызывается какая-то другая функция.

Этот метод имеет побочное полезное свойство: проверяются ошибки, возвращаемые функциями, код возврата которых обычно игнорируется, например `close` и `pthread_mutex_lock`.

Далее в тексте книги мы будем использовать эти функции-обертки, если только не потребуется явно проверить наличие ошибки и обработать ее произвольным образом, отличным от завершения процесса. Мы не приводим в книге исходный код для всех оберток, но он свободно доступен в Интернете (см. предисловие).

Значение errno

При возникновении ошибки в функции Unix глобальной переменной errno присваивается положительное значение, указывающее на тип ошибки; при этом функция обычно возвращает значение -1. Наша функция err_sys выводит соответствующее коду ошибки сообщение (например, Resource temporarily unavailable — ресурс временно недоступен, — если переменная errno имеет значение EAGAIN).

Функция присваивает значение переменной errno только при возникновении ошибки. В случае нормального завершения работы значение этой переменной не определено. Все положительные значения соответствуют константам с именами из заглавных букв, начинающимися с E, определяемым обычно в заголовочном файле <sys/errno.h>. Отсутствию ошибок соответствует значение 0.

При работе с несколькими потоками в каждом из них должна быть собственная переменная errno. Выделение переменной каждому потоку происходит автоматически, однако обычно это требует указания компилятору на то, что должна быть возможность повторного входа в программу. Задается это с помощью ключей -D_REENTRANT или -D_POSIX_C_SOURCE=199506L или аналогичных. Часто в заголовке <errno.h> переменная errno определяется как макрос, раскрываемый в вызов функции, если определена константа _REENTRANT. Функция обеспечивает доступ к копии errno, относящейся к данному потоку.

Далее в тексте мы используем выражения наподобие «функция mq_send возвращает EMSGSIZE», означающие, что функция возвращает ошибку (обычно возвращаемое значение при этом равно -1) и присваивает переменной errno значение указанной константы.

В настоящее время стандарты Unix определяются Posix и The Open Group.

Posix

Название Posix образовано от «Portable Operating System Interface», что означает приблизительно «интерфейс переносимых операционных систем». Это не один стандарт, а целое семейство, разработанное Институтом инженеров по электротехнике и радиоэлектронике (Institute for Electrical and Electronics Engineers — IEEE). Стандарты Posix были также приняты в качестве международных стандартов ISO (International Organization for Standardization, Международная организация по стандартизации) и IEC (International Electrotechnical Commission, Международная электротехническая комиссия), или ISO/IEC. Стандарты Posix прошли несколько стадий разработки.

■ Стандарт IEEE 1003.1-1988 (317 страниц) был первым стандартом Posix. Он определял интерфейс взаимодействия языка C с ядром Unix-типа в следующих областях: примитивы для реализации процессов (вызовы fork, exec, сигналы и таймеры), среда процесса (идентификаторы пользователей, группы процессов), файлы и каталоги (все функции ввода-вывода), работа с терминалом, базы данных системы (файлы паролей и групп), форматы архивов tar и cpio.

ПРИМЕЧАНИЕ

Первый стандарт Posix вышел в рабочем варианте под названием IEEEIX в 1986 году. Название Posix было предложено Ричардом Штолманом (Richard Stallman).

■ Затем вышел стандарт IEEE 1003.1-1990 (356 страниц). Он одновременно являлся и международным стандартом ISO/IEC 9945-1:1990. По сравнению с версией 1988 года изменения в версии 1990 года были минимальными. К заголовку было добавлено: «Part 1: System Application Program Interface (API) [C Language]» («Часть 1: Системный интерфейс разработки программ (API) [Язык C]»), и это означало, что стандарт описывал программный интерфейс (API) языка C.

■ IEEE 1003.2-1992 вышел в двух томах общим объемом около 1300 страниц, и его заголовок содержал строку «Part 2: Shell and Utilities» (Часть 2: «Интерпретатор и утилиты»). Эта часть определяла интерпретатор (основанный на Bourne shell в Unix System V) и около ста утилит (программ, обычно вызываемых из интерпретатора — от awk и basename до vi и yacc). В настоящей книге мы будем ссылаться на этот стандарт под именем Posix. 2.

■ IEEE 1003.1b-1993 (590 страниц) изначально был известен как IEEE P1003.4. Этот стандарт представлял собой дополнение к стандарту 1003.1-1990 и включал расширения реального времени, разработанные рабочей группой P1003.4: синхронизацию файлов, асинхронный ввод-вывод, семафоры, управление памятью, планирование выполнения (scheduling), часы, таймеры и очереди сообщений.

■ IEEE 1003.1, издание 1996 года [8] (743 страницы), включает 1003.1-1990 (базовый интерфейс API), 1003.1b-1993 (расширения реального времени), 1003.1-1995 (Pthreads — программные потоки Posix) и 1003.1i-1995 (технические поправки к 1003.1b). Этот стандарт также называется ISO/IEC 9945-1: 1996. В него были добавлены три главы о потоках и дополнительные разделы, касающиеся синхронизации потоков (взаимные исключения и условные переменные), планирование выполнения потоков, планирование синхронизации. В настоящей книге мы называем этот стандарт Posix.1.

ПРИМЕЧАНИЕ

Более четверти из 743 страниц стандарта представляли собой приложение, озаглавленное «Rationale and Notes» («Обоснование и примечания»). Это обоснование содержит историческую информацию и объяснение причин, по которым некоторые функции были или не были включены в стандарт. Часто обоснование оказывается не менее полезным, чем собственно стандарт.

К сожалению, стандарты IEEE не являются свободно доступными через Интернет. Информация о том, где можно заказать книгу, дана в библиографии под ссылкой [8]. Обратите внимание, что семафоры были определены в стандарте расширений реального времени, отдельно от взаимных исключений и условных переменных (которые были определены в стандарте Pthreads), что объясняет некоторые различия в интерфейсах API этих средств.

Наконец, заметим, что блокировки чтения-записи не являются частью стандартов Posix. Об этом более подробно рассказано в главе 8.

В будущем планируется выход новой версии IEEE 1003.1, включающей стандарт P1003.1g, сетевые интерфейсы (сокеты и XTI), которые описаны в первом томе этой книги.

В предисловии стандарта Posix.1 1996 года утверждается, что стандарт ISO/IEC 9945 состоит из следующих частей:

1. Системный интерфейс разработки программ (API) (язык C).

2. Интерпретатор и утилиты.

3. Администрирование системы (в разработке).

Части 1 и 2 представляют собой то, что мы называем Posix.1 и Posix.2.

Работа над стандартами Posix постоянно продолжается, и авторам книг, с ними связанных, приходится заниматься стрельбой по движущейся мишени. О текущем состоянии стандартов можно узнать на сайте <http://www.pasc.org/standing/sd11.html>.

The Open Group

The Open Group (Открытая группа) была сформирована в 1996 году объединением X/Open Company (основана в 1984 году) и Open Software Foundation (OSF, основан в 1988 году). Эта группа представляет собой международный консорциум производителей и потребителей из промышленности, правительства и образовательных учреждений. Их стандарты тоже выходили в нескольких версиях:

- В 1989 году X/Open опубликовала 3-й выпуск X/Open Portability Guide (Руководство по разработке переносимых программ) — XPG3.
- В 1992 году был опубликован четвертый выпуск (Issue 4), а в 1994 году — вторая его версия (Issue 4, Version 2). Последняя известна также под названием Spec 1170, где магическое число 1170 представляет собой сумму количества интерфейсов системы (926), заголовков (70) и команд (174). Есть и еще два названия: X/Open Single Unix Specification (Единая спецификация Unix) и Unix 95.
- В марте 1997 года было объявлено о выходе второй версии Единой спецификации Unix. Этот стандарт программного обеспечения называется также Unix 98, и именно так мы ссылаемся на эту спецификацию далее в тексте книги. Количество интерфейсов в Unix 98 возросло с 1170 до 1434, хотя для рабочей станции это количество достигает 3030, поскольку в это число включается CDE (Common Desktop Environment — общее окружение рабочего стола), которое, в свою очередь, требует системы X Window System и пользовательского интерфейса Motif. Подробно об этом написано в книге [9]. Полезную информацию можно также найти по адресу <http://www.UNIX-systems.org/version2>.

ПРИМЕЧАНИЕ

С этого сайта можно свободно скачать единую спецификацию Unix практически целиком.

Версии Unix и переносимость

Практически все версии Unix, с которыми можно столкнуться сегодня, соответствуют какому-либо варианту стандарта Posix.1 или Posix.2. Мы говорим «какому-либо», потому что после внесения изменений в Posix (например, Добавление расширений реального времени в 1993 и потоков в 1996) производителям обычно требуется год или два, чтобы подогнать свои программы под эти стандарты.

Исторически большинство систем Unix являются потомками либо BSD, либо System V, но различия между ними постепенно стираются, по мере того как производители переходят к использованию стандартов Posix. Основные различия лежат в области системного администрирования, поскольку ни один стандарт Posix на данный момент не описывает эту область.

В большинстве примеров этой книги мы использовали операционные системы Solaris 2.6 и Digital Unix 4.0B. Дело в том, что на момент написания книги (конец 1997 — начало 1998 года) только эти две операционные системы поддерживали System V IPC, Posix IPC и программные потоки Posix (Pthreads).

1.8. Комментарий к примерам IPC

Чаще всего для иллюстрации различных функций в книге используются три шаблона (модели) взаимодействия:

1. Сервер файлов: приложение клиент-сервер, причем клиент посыпает серверу запрос с именем файла, а сервер возвращает клиенту его содержимое.
2. Производитель-потребитель: один или несколько потоков или процессов (производителей) помещают данные в буфер общего пользования, а другие потоки или процессы (потребители) производят с этими данными различные операции.
3. Увеличение последовательного номера: один или несколько потоков или процессов увеличивают общий для всех индекс. Число это может храниться в файле с общим доступом или в совместно используемой области памяти.

Первый пример иллюстрирует различные формы передачи сообщений, а других два — разнообразные виды синхронизации и использования разделяемой памяти.

Таблицы 1.5, 1.6 и 1.7 представляют собой своего рода путеводитель по разрабатываемым нами программам на различные темы, изложенные в книге. В этих таблицах кратко описаны сами программы и указаны номера соответствующих листингов.

1.9. Резюме

Взаимодействие процессов традиционно является одной из проблемных областей в Unix. По мере развития системы предлагались различные решения, и ни одно из них не было совершенным. Мы подразделяем IPC на четыре главных типа.

1. Передача сообщений (каналы, FIFO, очереди сообщений).
2. Синхронизация (взаимные исключения, условные переменные, блокировки чтения-записи, семафоры).
3. Разделяемая память (неименованная и именованная).
4. Вызов процедур (двери в Solaris, RPC Sun).

Мы рассматриваем взаимодействие как отдельных потоков одного процесса, так и нескольких независимых процессов.

Живучесть каждого типа IPC определяется либо процессом, либо ядром, либо файловой системой в зависимости от продолжительности его существования. При выборе типа IPC для конкретного применения нужно учитывать его живучесть.

Другим свойством каждого типа IPC является пространство имен, определяющее идентификацию объектов IPC процессами и потоками, использующими его. Некоторые объекты не имеют имен (каналы, взаимные исключения, условные переменные, блокировки чтения-записи), другие обладают именами в рамках файловой системы (каналы FIFO), третьи характеризуются тем, что в главе 2 названо «именами IPC стандарта Posix», а четвертые — еще одним типом имен, который описан в главе 3 (ключи или идентификаторы IPC стандарта System V). Обычно сервер создает объект IPC с некоторым именем, а клиенты используют это имя для получения доступа к объекту.

В исходных кодах, приведенных в книге, используются функции-обертки, описанные в разделе 1.6, позволяющие уменьшить объем кода, обеспечивая, тем не менее, проверку возврата ошибки для любой вызываемой функции. Имена всех функций-оберток начинаются с заглавной буквы.

Стандарты IEEE Posix — Posix.1, определяющий основы интерфейса C в Unix, и Posix.2, определяющий основные команды, — это те стандарты, к которым движутся большинство производителей. Однако стандарты Posix в настоящее время быстро поглощаются (включаются в качестве части) и расширяются коммерческими стандартами, в частности The Open Group (Unix 98).

Таблица 1.5. Версии модели клиент-сервер

Таблица 1.6. Версии модели производитель-потребитель

Таблица 1.7. Версии программы с увеличением последовательного номера

Упражнения

1. На рис 1.1 изображены два процесса, обращающиеся к одному файлу. Если оба процесса только дописывают данные к концу файла (возможно, длинного), какой нужен будет тип синхронизации?
2. Изучите заголовочный файл <errno.h> в вашей системе и выясните, как определена errno.
3. Дополните табл. 1.3 используемыми вами функциями, поддерживаемыми Unix-системами.

2.1. Введение

Из имеющихся типов IPC следующие три могут быть отнесены к Posix IPC, то есть к методам взаимодействия процессов, соответствующим стандарту Posix:

- очереди сообщений Posix (глава 5);
- семафоры Posix (глава 10);
- разделяемая память Posix (глава 13).

Эти три вида IPC обладают общими свойствами, и для работы с ними используются похожие функции. В этой главе речь пойдет об общих требованиях к полным именам файлов, используемых в качестве идентификаторов, о флагах, указываемых при открытии или создании объектов IPC, и о разрешениях на доступ к ним.

Полный список функций, используемых для работы с данными типами IPC, приведен в табл. 2.1.

Таблица 2.1. Функции Posix IPC

В табл. 1.2 мы отметили, что три типа IPC стандарта Posix имеют идентификаторы (имена), соответствующие этому стандарту. Имя IPC передается в качестве первого аргумента одной из трех функций: `mq_open`, `sem_open` и `shm_open`, причем оно не обязательно должно соответствовать реальному файлу в файловой системе. Стандарт Posix.1 накладывает на имена IPC следующие ограничения:

- Имя должно соответствовать существующим требованиям к именам файлов (не превышать в длину `PATHMAX` байтов, включая завершающий символ с кодом 0).
- Если имя начинается со слэша (/), вызов любой из этих функций приведет к обращению к одной и той же очереди. В противном случае результат зависит от реализации.
- Интерпретация дополнительных слэшей в имени зависит от реализации.

Таким образом, для лучшей переносимости имена должны начинаться со слэша (/) и не содержать в себе дополнительных слэшей. К сожалению, эти правила, в свою очередь, приводят к проблемам с переносимостью.

В системе Solaris 2.6 требуется наличие начального слэша и запрещается использование дополнительных. Для очереди сообщений, например, при этом создаются три файла в каталоге `/tmp`, причем имена этих файлов начинаются с `.MQ`. Например, если аргумент функции `mq_open` имеет вид `/queue.1234`, то созданные файлы будут иметь имена `/tmp/.MQDqueue.1234`, `/tmp/.MQLqueue.1234` и `/tmp/.MQPqueue.1234`. В то же время в системе Digital Unix 4.0B просто создается файл с указанным при вызове функции именем.

Проблема с переносимостью возникает при указании имени с единственным слэшем в начале: при этом нам нужно иметь разрешение на запись в корневой каталог. Например, очередь с именем `/tmp.1234` допустима стандартом Posix и не вызовет проблем в системе Solaris, но в Digital Unix возникнет ошибка создания файла, если разрешения на запись в корневой каталогу программы нет. Если мы укажем имя `/tmp/test.1234`, проблемы в Digital Unix и аналогичных системах, создающих файл с указанным именем, пропадут (предполагается существование каталога `/tmp` и наличие у программы разрешения на запись в него, что обычно для большинства систем Unix), однако в Solaris использование этого имени будет невозможно.

Для решения подобных проблем с переносимостью следует определять имя в заголовке с помощью директивы `#define`, чтобы обеспечить легкость его изменения при переносе программы в другую систему.

ПРИМЕЧАНИЕ

Разработчики стремились разрешить использование очередей сообщений, семафоров и разделяемой памяти для существующих ядер Unix и в независимых бездисковых системах. Это тот случай, когда стандарт получается чересчур общим и в результате вызывает проблемы с переносимостью. В отношении Posix это называется «как стандарт становится нестандартным».

Стандарт Posix.1 определяет три макроса:

которые принимают единственный аргумент — указатель на структуру типа `stat`, содержимое которой задается функциями `fstat`, `lstat` и `stat`. Эти три макроса возвращают ненулевое значение, если указанный объект IPC (очередь сообщений, семафор или сегмент разделяемой памяти) реализован как особый вид файла и структура `stat` ссылается на этот тип. В противном случае макрос возвращает 0.

ПРИМЕЧАНИЕ

К сожалению, проку от этих макросов мало, потому что нет никаких гарантий, что эти типы IPC реализованы как отдельные виды файлов. Например, в Solaris 2.6 все три макроса всегда возвращают 0.

Все прочие макросы, используемые для проверки типа файла, имеют имена, начинающиеся с `S_IS`, и принимают всегда единственный аргумент: поле `st_mode` структуры `stat`. Поскольку макросы, используемые для проверки типа IPC, принимают аргументы другого типа, их имена начинаются с `S_TYPEIS`.

Функция px_ipc_name

Существует и другое решение упомянутой проблемы с переносимостью. Можно определить нашу собственную функцию px_ipc_name, которая добавляет требуемый каталог в качестве префикса к имени Posix IPC.

ПРИМЕЧАНИЕ

Так выглядят листинги наших собственных функций, то есть функций, не являющихся стандартными системными. Обычно включается заголовочный файл uprisc.h (листинг B.1).

Аргумент *name* (имя) не должен содержать слэшей. Тогда, например, при вызове px_ipc_name("test1") будет возвращен указатель на строку /test1 в Solaris 2.6 или на строку /tmp/test1 в Digital Unix 4.0B. Память для возвращаемой строки выделяется динамически и освобождается вызовом free. Можно установить произвольное значение переменной окружения PX_IPC_NAME, чтобы задать другой каталог по умолчанию.

В листинге 2.1 приведен наш вариант реализации этой функции.

ПРИМЕЧАНИЕ

Возможно, в этом листинге вы в первый раз встретитесь с функцией snprintf. Значительная часть существующих программ используют вместо нее функцию sprintf, однако последняя не производит проверки переполнения приемного буфера. В отличие от нее snprintf получает в качестве второго аргумента размер приемного буфера и впоследствии предотвращает его переполнение.

Умышленное переполнение буфера программы, использующей sprintf, в течение многих лет использовалось хакерами для взлома систем.

Функция snprintf еще не является частью стандарта ANSI C, но планируется ее включение в обновленный стандарт, называющийся C9X. Тем не менее многие производители включают ее в стандартную библиотеку C. Везде в тексте мы используем функцию snprintf в нашем собственном варианте, обеспечивающем вызов sprintf, если в системной библиотеке функция snprintf отсутствует.

2.3. Создание и открытие каналов IPC

Все три функции, используемые для создания или открытия объектов IPC: `mq_open`, `sem_open` и `shm_open`, — принимают специальный флаг `oflag` в качестве второго аргумента. Он определяет параметры открытия запрашиваемого объекта аналогично второму аргументу стандартной функции `open`. Все константы, из которых можно формировать этот аргумент, приведены в табл. 2.2.

Таблица 2.2. Константы, используемые при создании и открытии объектов IPC

Первые три строки описывают тип доступа к создаваемому объекту: только чтение, только запись, чтение и запись. Очередь сообщений может быть открыта в любом из трех режимов доступа, тогда как для семафора указание этих констант не требуется (для любой операции с семафором требуется доступ на чтение и запись). Наконец, объект разделяемой памяти не может быть открыт только на запись.

Указание прочих флагов из табл. 2.2 не является обязательным.

`O_CREAT` — создание очереди сообщений, семафора или сегмента разделяемой памяти, если таковой еще не существует (см. также флаг `O_EXCL`, влияющий на результат).

При создании новой очереди сообщений, семафора или сегмента разделяемой памяти требуется указание по крайней мере одного дополнительного аргумента, определяющего режим. Этот аргумент указывает биты разрешения на доступ к файлу и формируется путем побитового логического сложения констант из табл. 2.3.

Таблица 2.3. Константы режима доступа при создании нового объекта IPC

Эти константы определены в заголовке `<sys/stat.h>`. Указанные биты разрешений изменяются наложением маски режима создания файлов для данного процесса (с. 83-85 [21]) или с помощью команды интерпретатора `umask`.

Как и со вновь созданным файлом, при создании очереди сообщений, семафора или сегмента разделяемой памяти им присваивается идентификатор пользователя, соответствующий действующему (effective) идентификатору пользователя процесса. Идентификатор группы семафора или сегмента разделяемой памяти устанавливается равным действующему групповому идентификатору процесса или групповому идентификатору, установленному по умолчанию для системы в целом. Групповой идентификатор очереди сообщений всегда устанавливается равным действующему групповому идентификатору процесса (на с. 77-78 [21] рассказывается о групповых и пользовательских идентификаторах более подробно).

ПРИМЕЧАНИЕ

Кажется странным наличие разницы в установке группового идентификатора для разных видов Posix IPC. Групповой идентификатор нового файла, создаваемого с помощью функции `open`, устанавливается равным либо действительному идентификатору группы процесса, либо идентификатору группы каталога, в котором создается файл, но функции IPC не могут заранее предполагать, что для объекта IPC создается реальный файл в файловой системе.

`O_EXCL` — если этот флаг указан одновременно с `O_CREAT`, функция создает новую очередь сообщений, семафор или объект разделяемой памяти только в том случае, если таковой не существует. Если объект уже существует и указаны флаги `O_CREAT | O_EXCL`, возвращается ошибка `EEXIST`.

Проверка существования очереди сообщений, семафора или сегмента разделяемой памяти и его создание (в случае отсутствия) должны производиться только одним процессом. Два аналогичных флага имеются и в System V IPC, они описаны в разделе 3.4.

`O_NONBLOCK` — этот флаг создает очередь сообщений без блокировки. Блокировка обычно устанавливается для считывания из пустой очереди или записи в полную очередь. Об этом более подробно рассказано в подразделах, посвященных функциям `mq_send` и `mq_receive` раздела 5.4.

`O_TRUNC` — если уже существующий сегмент общей памяти открыт на чтение и запись, этот флаг указывает на необходимость сократить его размер до 0.

На рис. 2.1 показана реальная последовательность логических операций при открытии объекта IPC. Что именно подразумевается под проверкой разрешений доступа, вы узнаете в разделе 2.4. Другой подход к изображенному на рис. 2.1 представлен в табл. 2.4.



Рис. 2.1. Логика открытия объекта IPC

Обратите внимание, что в средней строке табл. 2.4, где задан только флаг O_CREAT, мы не получаем никакой информации о том, был ли создан новый объект или открыт существующий.

Таблица 2.4. Логика открытия объекта IPC

2.4. Разрешения IPC

Новая очередь сообщений, именованный семафор или сегмент разделяемой памяти создается функциями `mq_open`, `sem_open` и `shm_open`, при условии, что аргумент `oflag` содержит константу `O_CREAT`. Согласно табл. 2.3, любому из данных типов IPC присваиваются определенные права доступа (permissions), аналогичные разрешениям доступа к файлам в Unix.

При открытии существующей очереди сообщений, семафора или сегмента разделяемой памяти теми же функциями (в случае, когда не указан флаг `O_CREAT` или указан `O_CREAT` без `O_EXCL` и объект уже существует) производится проверка разрешений:

1. Проверяются биты разрешений, присвоенные объекту IPC при создании.
2. Проверяется запрошенный тип доступа (`O_RDONLY`, `O_WRONLY`, `O_RDWR`).
3. Проверяется действующий идентификатор пользователя вызывающего процесса, действующий групповой идентификатор процесса и дополнительные групповые идентификаторы процесса (последние могут не поддерживаться).

Большинством систем Unix производятся следующие конкретные проверки:

1. Если действующий идентификатор пользователя для процесса есть 0 (привилегированный пользователь), доступ будет разрешен.
2. Если действующий идентификатор пользователя процесса совпадает с идентификатором владельца объекта IPC: если соответствующий бит разрешения для пользователя установлен, доступ разрешен, иначе в доступе отказывается.
- Под соответствующим битом разрешения мы подразумеваем, например, бит разрешения на чтение, если процесс открывает объект только для чтения. Если процесс открывает объект для записи, должен быть установлен соответственно бит разрешения на запись для владельца (user-write).
3. Если действующий идентификатор группы процесса или один из дополнительных групповых идентификаторов процесса совпадает с групповым идентификатором объекта IPC: если соответствующий бит разрешения для группы установлен, доступ будет разрешен, иначе в доступе отказывается.
4. Если соответствующий бит разрешения доступа для прочих пользователей установлен, доступ будет разрешен, иначе в доступе будет отказано.

Эти четыре проверки производятся в указанном порядке. Следовательно, если процесс является владельцем объекта IPC (шаг 2), доступ разрешается или запрещается на основе одних только разрешений пользователя (владельца). Разрешения группы при этом не проверяются. Аналогично, если процесс не является владельцем объекта IPC, но принадлежит к нужной группе, доступ разрешается или запрещается на основе разрешений группы — разрешения для прочих пользователей при этом не проверяются.

ПРИМЕЧАНИЕ

Согласно табл. 2.2, функция `sem_open` не использует флаги `O_RDONLY`, `O_WRONLY`, `O_RDWR`. В разделе 10.2, однако, будет сказано о том, что некоторые реализации Unix подразумевают наличие флага `O_RDWR`, потому что любое обращение к семафору подразумевает чтение и запись его значения.

2.5. Резюме

Три типа Posix IPC — очереди сообщений, семафоры и разделяемая память — идентифицируются их полными именами. Они могут являться или не являться реальными именами файлов в файловой системе, и это вызывает проблемы с переносимостью. Решение проблемы — использовать собственную функцию `px_ipc_name`. При создании или открытии объекта IPC требуется указать набор флагов, аналогичных указываемым при использовании функции `open`. При создании нового объекта IPC требуется указать разрешения для него, используя те же константы `S_xxx`, что и для функции `open` (табл. 2.3). При открытии существующего объекта IPC производится проверка разрешений процесса, аналогичная проверке при открытии файла.

Упражнения

1. Каким образом биты установки идентификатора пользователя (set-user-ID, SUID) и установки идентификатора группы (set-group-ID) (раздел 4.4 [21]) программы, использующей Posix IPC, влияют на проверку разрешений, описанную в разделе 2.4?
2. Когда программа открывает объект IPC, как она может определить, был ли создан новый объект IPC или производится обращение к существующему объекту?

3.1. Введение

Из имеющихся типов IPC следующие три могут быть отнесены к System V IPC, то есть к методам взаимодействия процессов, соответствующим стандарту System V:

- очереди сообщений System V (глава 6);
- семафоры System V (глава 11);
- общая память System V (глава 14).

Термин «System V IPC» говорит о происхождении этих средств: впервые они появились в Unix System V. У них много общего: схожи функции, с помощью которых организуется доступ к объектам; также схожи формы хранения информации в ядре. В этой главе описываются общие для трех типов IPC черты.

Информация о функциях сведена в табл. 3.1.

Таблица 3.1. Функции System V IPC

ПРИМЕЧАНИЕ

Информация об истории разработки и развития функций System V IPC не слишком легко доступна. [16] предоставляет следующую информацию: очереди сообщений, семафоры и разделяемая память этого типа были разработаны в конце 70-х в одном из филиалов Bell Laboratories в городе Колумбус, штат Огайо, для одной из версий Unix, предназначеннной для внутреннего использования. Версия эта называлась Columbus Unix, или CB Unix. Она использовалась в так называемых системах поддержки операций — системах обработки транзакций — для автоматизации управления и ведения записей в телефонной компании. System V IPC были добавлены в коммерческую версию Unix System V. приблизительно в 1983 году.

В табл. 1.2 было отмечено, что в именах трех типов System V IPC использовались значения `key_t`. Заголовочный файл `<sys/types.h>` определяет тип `key_t` как целое (по меньшей мере 32-разрядное). Значения переменным этого типа обычно присваиваются функцией `ftok`.

Функция `ftok` преобразовывает существующее полное имя и целочисленный идентификатор в значение типа `key_t` (называемое ключом IPC — IPC key):

На самом деле функция использует полное имя файла и младшие 8 бит идентификатора для формирования целочисленного ключа IPC.

Эта функция действует в предположении, что для конкретного приложения, использующего IPC, клиент и сервер используют одно и то же полное имя объекта IPC, имеющее какое-то значение в контексте приложения. Это может быть имя демона сервера или имя файла данных, используемого сервером, или имя еще какого-нибудь объекта файловой системы. Если клиенту и серверу для связи требуется только один канал IPC, идентификатору можно присвоить, например, значение 1. Если требуется несколько каналов IPC (например, один от сервера к клиенту и один в обратную сторону), идентификаторы должны иметь разные значения: например, 1 и 2. После того как клиент и сервер договорятся о полном имени и идентификаторе, они оба вызывают функцию `ftok` для получения одинакового ключа IPC.

Большинство реализаций `ftok` вызывают функцию `stat`, а затем объединяют:

- информацию о файловой системе, к которой относится полное имя *pathname* (поле `st_dev` структуры `stat`);
- номер узла (i-node) в файловой системе (поле `st_ino` структуры `stat`);
- младшие 8 бит идентификатора (который не должен равняться нулю).

Из комбинации этих трех значений обычно получается 32-разрядный ключ. Нет никакой гарантии того, что для двух различных путей с одним и тем же идентификатором получатся разные ключи, поскольку количество бит информации в трех перечисленных элементах (идентификатор файловой системы, номер узла, идентификатор IPC) может превышать число бит в целом (см. упражнение 3.5).

ПРИМЕЧАНИЕ

Номер узла всегда отличен от нуля, поэтому большинство реализаций определяют константу `IPC_PRIVATE` (раздел 3.4) равной нулю.

Если указанное полное имя не существует или недоступно вызывающему процессу, `ftok` возвращает значение `-1`. Помните, что файл, имя которого используется для вычисления ключа, не должен быть одним из тех, которые создаются и удаляются сервером в процессе работы, поскольку каждый раз при создании заново эти файлы получают, вообще говоря, другой номер узла, а это может изменить ключ, возвращаемый функцией `ftok` при очередном вызове.

Пример

Программа в листинге 3.1 принимает полное имя в качестве аргумента командной строки, вызывает функции stat и ftok, затем выводит значения полей st_dev и st_ino структуры stat и получающийся ключ IPC. Эти три значения выводятся в шестнадцатеричном формате, поэтому легко видеть, как именно ключ IPC формируется из этих двух значений и идентификатора 0x57.

Выполнение этой программы в системе Solaris 2.6 приведет к следующим результатам:

Очевидно, идентификатор определяет старшие 8 бит ключа; младшие 12 бит st_dev определяют следующие 12 бит ключа, и наконец, младшие 12 бит st_ino определяют младшие 12 бит ключа.

Цель этого примера не в том, чтобы впоследствии рассчитывать на такой способ формирования ключа из перечисленной информации, а в том, чтобы проиллюстрировать алгоритм комбинации полного имени и идентификатора конкретной реализацией. В других реализациях алгоритм может быть другим.

ПРИМЕЧАНИЕ

В FreeBSD используются младшие 8 бит идентификатора, младшие 8 бит st_dev и младшие 16 бит st_ino.

Учтите, что отображение, производимое функцией ftok, — одностороннее, поскольку часть бит st_dev и st_ino не используются. По данному ключу нельзя определить полное имя файла, заданное для вычислений.

3.3. Структура ipc_perm

Для каждого объекта IPC, как для обычного файла, в ядре хранится набор информации, объединенной в структуру.

Эта структура вместе с другими переименованными константами для функций System V IPC определена в файле `<sys/ipc.h>`. В этой главе мы расскажем о полях структуры `ipc_perm` более подробно.

3.4. Создание и открытие каналов IPC

Три функции `getXXX`, используемые для создания или открытия объектов IPC (табл. 3.1), принимают ключ IPC (типа `key_t`) в качестве одного из аргументов и возвращают целочисленный идентификатор. Этот идентификатор отличается от того, который передавался функции `ftok`, как мы вскоре увидим. У приложения есть две возможности задания ключа (первого аргумента функций `getXXX`):

1. Вызвать `ftok`, передать ей полное имя и идентификатор.
2. Указать в качестве ключа константу `IPCPRIVATE`, гарантирующую создание нового уникального объекта IPC.

Последовательность действий иллюстрирует рис. 3.1.



Рис. 3.1. Вычисление идентификаторов IPC по ключам

Все три функции `getXXX` (табл. 3.1) принимают в качестве второго аргумента набор флагов `oflag`, задающий биты разрешений чтения-записи (поле `mode` структуры `ipc_perm`) для объекта IPC и определяющий, создается ли новый объект IPC или производится обращение к уже существующему. Для этого имеются следующие правила.

- Ключ `IPC_PRIVATE` гарантирует создание уникального объекта IPC. Никакие возможные комбинации полного имени и идентификатора не могут привести к тому, что функция `ftok` вернет в качестве ключа значение `IPC_PRIVATE`.
- Установка бита `IPC_CREAT` аргумента `oflag` приводит к созданию новой записи для указанного ключа, если она еще не существует. Если же обнаруживается существующая запись, возвращается ее идентификатор.

Одновременная установка битов `IPC_CREAT` и `IPC_EXCL` аргумента `oflag` приводит к созданию новой записи для указанного ключа только в том случае, если такая запись еще не существует. Если же обнаруживается существующая запись, функция возвращает ошибку `EEXIST` (объект IPC уже существует).

Комбинация `IPC_CREAT` и `IPC_EXCL` в отношении объектов IPC действует аналогично комбинации `O_CREAT` и `O_EXCL` для функции `open`.

Установка только бита `IPC_EXCL` без `IPC_CREAT` никакого эффекта не дает.

Логическая диаграмма последовательности действий при открытии объекта IPC изображена на рис. 3.2. В табл. 3.2 показан альтернативный взгляд на этот процесс.



Рис. 3.2. Диаграмма открытия объекта IPC

Обратите внимание, что в средней строке табл. 3.2 для флага `IPC_CREAT` без `IPC_EXCL` мы не получаем никакой информации о том, был ли создан новый объект или получен доступ к существующему. Для большинства приложений характерно создание сервером объекта IPC с указанием `IPC_CREAT` (если безразлично, существует ли уже объект) или `IPC_CREAT | IPC_EXCL` (если требуется проверка существования объекта). Клиент вообще не указывает флагов, предполагая, что сервер уже создал объект.

ПРИМЕЧАНИЕ

Функции System V IPC в отличие от функций Posix IPC определяют свои собственные константы `IPC_xxx` вместо использования `O_CREAT` и `OEXCL`, принимаемых стандартной функцией `open` (табл. 2.2).

Обратите также внимание на то, что функции System V IPC совмещают константы `IPC_xxx` с битами разрешений (описанными в следующем разделе) в едином аргументе `oflag`, тогда как для функции `open` и для Posix IPC характерно наличие двух аргументов: `oflag`, в котором задаются флаги вида `O_xxx`, и `mode`, определяющего биты разрешений доступа.

Таблица 3.2. Логика создания и открытия объектов IPC

3.5. Разрешения IPC

При создании нового объекта IPC с помощью одной из функций `getXXX`, вызванной с флагом `IPC_CREAT`, в структуру `ipc_perm` заносится следующая информация (раздел 3.3):

1. Часть битов аргумента `oflag` задают значение поля `mode` структуры `ipc_perm`. В табл. 3.3 приведены биты разрешений для трех типов IPC (запись `>>3` означает сдвиг вправо на три бита).
2. Поля `cuid` и `cgid` получают значения, равные действующим идентификаторам пользователя и группы вызывающего процесса. Эти два поля называются идентификаторами создателя.
3. Поля `uid` и `gid` структуры `ipc_perm` также устанавливаются равными действующим идентификаторам вызывающего процесса. Эти два поля называются идентификаторами владельца.

Таблица 3.3. Значения `mode` для разрешений чтения-записи IPC

Идентификатор создателя изменяться не может, тогда как идентификатор владельца может быть изменен процессом с помощью вызова функции `ctlXXX` для данного механизма IPC с командой `IPC_SET`. Три функции `ctlXXX` позволяют процессу изменять биты разрешений доступа (поле `mode`) объекта IPC.

ПРИМЕЧАНИЕ

В большинстве реализаций определены шесть констант: `MSG_R`, `MSG_W`, `SEM_R`, `SEM_A`, `SHM_R` и `SHM_W`, показанные в табл. 3.3. Константы эти определяются в заголовочных файлах `<sys/msg.h>`, `<sys/sem.h>` и `<sys/shm.h>`. Однако стандарт Unix 98 не требует их наличия. Суффикс `A` в `SEM_A` означает «`alter`» (изменение).

Тройка функций `getXXX` не используют стандартную маску создания файла Unix. Разрешения очереди сообщений, семафора и разделяемой памяти устанавливаются в точности равными аргументу функции.

Posix IPC не дает создателю IPC возможности изменить владельца объекта. В Posix нет аналогов команды `IPC_SET`. Однако в Posix IPC имя объекта принадлежит файловой системе, и потому владелец может быть изменен привилегированным пользователем с помощью команды `chown`.

Когда какой-либо процесс предпринимает попытку доступа к объекту IPC, производится двухэтапная проверка: первый раз при открытии файла (функция `getXXX`) и затем каждый раз при обращении к объекту IPC:

1. При установке доступа к существующему объекту IPC с помощью одной из функций `getXXX` производится первичная проверка аргумента `oflag`, вызывающего функцию процесса. Аргумент не должен указывать биты доступа, не установленные в поле `mode` структуры `ipc_perm` (нижний квадрат на рис. 3.2). Например, процесс-сервер может установить значение члена `mode` для своей очереди входящих сообщений, сбросив биты чтения для группы и прочих пользователей. Любой процесс, попытавшийся указать эти биты в аргументе `oflag` функции `msgget`, получит ошибку. Надо отметить, что от этой проверки, производимой функциями `getXXX`, мало пользы. Она подразумевает наличие у вызывающего процесса информации о том, к какой группе пользователей он принадлежит: он может являться владельцем файла, может принадлежать к той же группе или к прочим пользователям. Если создающий процесс сбросит некоторые биты разрешений, а вызывающий процесс попытается их установить, функция `getXXX` вернет ошибку. Любой процесс может полностью пропустить эту проверку, указав аргумент `oflag`, равный 0, если заранее известно о существовании объекта IPC.

2. При любой операции с объектами IPC производится проверка разрешений для процесса, эту операцию запрашивающего. Например, каждый раз когда процесс пытается поместить сообщение в очередь с помощью команды `msgsnd`, производятся нижеследующие проверки (при получении доступа последующие этапы пропускаются).

Привилегированному пользователю доступ предоставляется всегда.

Если действующий идентификатор пользователя совпадает со значением `uid` или `cuid` объекта IPC и установлен соответствующий бит разрешения доступа в поле `mode` объекта IPC, доступ будет разрешен. Под соответствующим битом разрешения доступа подразумевается бит, разрешающий чтение, если вызывающий процесс запрашивает операцию чтения для данного объекта IPC (например, получение сообщения из очереди), или бит, разрешающий запись, если процесс хочет осуществить ее.

- Если действующий идентификатор группы совпадает со значением gid или cgid объекта IPC и установлен соответствующий бит разрешения доступа в поле mode объекта IPC, доступ будет разрешен.
- Если доступ не был разрешен на предыдущих этапах, проверяется наличие соответствующих установленных битов доступа для прочих пользователей.

3.6. Повторное использование идентификаторов

Структура `ipc_perm` (раздел 3.3) содержит переменную `seq`, в которой хранится порядковый номер канала. Эта переменная представляет собой счетчик, заводимый ядром для каждого объекта IPC в системе. При удалении объекта IPC номер канала увеличивается, а при переполнении сбрасывается в ноль.

ПРИМЕЧАНИЕ

В этом разделе мы описываем характерную для SVR4 реализацию. Стандарт Unix 98 не исключает использование других вариантов.

В счетчике возникает потребность по двум причинам. Во-первых, вспомним о дескрипторах файлов, хранящихся в ядре для каждого из открытых файлов. Они обычно представляют собой небольшие целые числа, имеющие значение только внутри одного процесса — для каждого процесса создаются собственные дескрипторы. Прочитать из файла с дескриптором 4 можно только в том процессе, в котором есть открытый файл с таким дескриптором. Есть ли открытые файлы с тем же дескриптором в других процессах — значения не имеет. В отличие от дескрипторов файлов идентификаторы System V IPC устанавливаются для всей системы, а не для процесса.

Идентификатор IPC возвращается одной из функций `getXXX`: `msgget`, `semget`, `shmget`. Как и дескрипторы файлов, идентификаторы представляют собой целые числа, имеющие в отличие от дескрипторов одинаковое значение для всех процессов. Если два неродственных процесса (клиент и сервер) используют одну очередь сообщений, ее идентификатор, возвращаемый функцией `msgget`, должен иметь одно и то же целочисленное значение в обоих процессах, чтобы они получили доступ к одной и той же очереди. Такая особенность дает возможность какому-либо процессу, созданному злоумышленником, попытаться прочесть сообщение из очереди, созданной другим приложением, последовательно перебирая различные идентификаторы (если бы они представляли собой небольшие целые числа) и надеясь на существование открытой в текущий момент очереди, доступной для чтения всем. Если бы идентификаторы представляли собой небольшие целые числа (как дескрипторы файлов), вероятность найти правильный идентификатор составляла бы около 1/50 (предполагая ограничение в 50 дескрипторов на процесс).

Для исключения такой возможности разработчики средств IPC решили расширить возможный диапазон значений идентификатора так, чтобы он включал вообще все целые числа, а не только небольшие. Расширение диапазона обеспечивается путем увеличения значения идентификатора, возвращаемого вызывающему процессу, на количество записей в системной таблице IPC каждый раз, когда происходит повторное использование одной из них. Например, если система настроена на использование не более 50 очередей сообщений, при первом использовании первой записи процессу будет возвращен идентификатор 0. После удаления этой очереди сообщений при попытке повторного использования первой записи в таблице процессу будет возвращен идентификатор 50. Далее он будет принимать значения 100, 150 и т. д. Поскольку `seq` обычно определяется как длинное целое без знака (`ulong` — см. структуру `ipc_perm` в разделе 3.3), возврат к уже использовавшимся идентификаторам происходит, когда запись в таблице будет использована 85899346 раз ($2^{32}/50$ в предположении, что целое является 32-разрядным).

Второй причиной, по которой понадобилось ввести последовательный номер канала, является необходимость исключить повторное использование идентификаторов System V IPC через небольшой срок. Это помогает гарантировать то, что досрочно завершивший работу и впоследствии перезапущенный сервер не станет использовать тот же идентификатор.

Иллюстрируя эту особенность, программа в листинге 3.2 выводит первые десять значений идентификаторов, возвращаемых `msgget`.

При очередном прохождении цикла `msgget` создает очередь сообщений, а `msgctl` с командой `IPC_RMID` в качестве аргумента удаляет ее. Константа `SVMSG_MODE` определяется в нашем заголовочном файле `uprisc.h` (листинг B.1) и задает разрешения по умолчанию для очереди сообщений System V. Вывод программы будет иметь следующий вид:

При повторном запуске программы мы увидим наглядную иллюстрацию того, что последовательный номер канала — это переменная, хранящаяся в ядре и продолжающая существовать и после завершения процесса.

3.7. Программы ipcs и ipcrm

Поскольку объектам System V IPC не сопоставляются имена в файловой системе, мы не можем просмотреть их список или удалить их, используя стандартные программы ls и rm. Вместо них в системах, поддерживающих эти типы IPC, предоставляются две специальные программы: ipcs, выводящая различную информацию о свойствах System V IPC, и ipcrm, удаляющая очередь сообщений System V, семафор или сегмент разделяемой памяти. Первая из этих функций поддерживает около десятка параметров командной строки, управляющих отображением информации о различных типах IPC. Второй (ipcrm) можно задать до шести параметров. Подробную информацию о них можно получить в справочной системе.

ПРИМЕЧАНИЕ

Поскольку System V IPC не стандартизуется Posix, эти команды не входят в Posix.2. Они описаны в стандарте Unix 98.

3.8. Ограничения ядра

Большинству реализаций System V IPC свойственно наличие внутренних ограничений, налагаемых ядром. Это, например, максимальное количество очередей сообщений или ограничение на максимальное количество семафоров в наборе. Характерные значения этих ограничений приведены в табл. 6.2, 11.1 и 14.1. Большая часть ограничений унаследована от исходной реализации System V.

ПРИМЕЧАНИЕ

Раздел 11.2 книги [1] и глава 8 [6] описывают реализацию очередей сообщений, семафоров и разделяемой памяти в System V. Некоторые из этих ограничений описаны уже там.

К сожалению, некоторые из накладываемых ограничений достаточно жестки, поскольку они унаследованы от исходной реализации, базировавшейся на системе с небольшим адресным пространством (16-разрядный PDP-11). К счастью, большинство систем позволяют администратору изменять некоторые из установленных по умолчанию ограничений, но необходимые для этого действия специфичны для каждой версии Unix. В большинстве случаев после внесения изменений требуется перезагрузка ядра. К сожалению, в некоторых реализациях для хранения некоторых параметров до сих пор используются 16-разрядные целые, а это уже устанавливает аппаратные ограничения.

В Solaris 2.6, например, таких ограничений 20. Их текущие значения можно вывести на экран, используя команду sysdef. Учтите, что вместо реальных значений будут выведены нули, если соответствующий модуль ядра не загружен (то есть средство не было ранее использовано). Это можно исключить, добавив к файлу /etc/system любой из нижеследующих операторов. Файл /etc/system считывается в процессе перезагрузки системы:

Последние шесть символов имени слева от знака равенства представляют собой переменные, перечисленные в табл. 6.2, 11.1 и 14.1.

В Digital Unix 4.0B программа sysconfig позволяет узнать или изменить множество параметров и ограничений ядра. Ниже приведен вывод этой команды, запущенной с параметром -q. Команда выводит текущие ограничения для подсистемы ipcs. Некоторые строки в выводе, не имеющие отношения к средствам IPC System V, были опущены:

Для этих параметров можно указать другие значения по умолчанию, изменив файл /etc/sysconfigtab. Делать это следует с помощью программы sysconfigdb. Этот файл также считывается в процессе начальной загрузки системы.

3.9. Резюме

Первым аргументом функций msgget, semget и shmget является ключ IPC System V. Эти ключи вычисляются по полному имени файла с помощью системной функции ftok. В качестве ключа можно также указать значение IPCPRIVATE. Эти три функции создают новый объект IPC или открывают существующий и возвращают идентификатор System V IPC — целое число, которое потом используется для распознавания объекта в прочих функциях, имеющих отношение к IPC. Эти идентификаторы имеют смысл не только в рамках одного процесса (как дескрипторы файлов), но и в рамках всей системы. Они могут повторно использоваться ядром, но лишь спустя некоторое время.

С каждым объектом System V IPC связана структура ipc_perm, содержащая информацию о нем, такую как идентификатор пользователя владельца, идентификатор группы, разрешения чтения и записи и др. Одним из отличий между System V и Posix IPC является то, что для объекта IPC System V эта информация доступна всегда (доступ к ней можно получить с помощью одной из функций XXXctl с аргументом IPC_STAT), а в Posix IPC доступ к ней зависит от реализации. Если объект Posix IPC хранится в файловой системе и мы знаем его имя в ней, мы можем получить доступ к этой информации, используя стандартные средства файловой системы.

При создании нового или открытии существующего объекта System V IPC функции getXXX передаются два флага (IPC_CREAT и IPC_EXCL) и 9 бит разрешений.

Без сомнения, главнейшей проблемой в использовании System V IPC является наличие искусственных ограничений в большинстве реализаций. Ограничения накладываются на размер объектов, причем они берут свое начало от самых первых реализаций. Это означает, что для интенсивного использования средств System V IPC приложениями требуется изменение ограничений ядра, а внесение этих изменений в каждой системе осуществляется по-разному.

Упражнения

1. Прочтайте о функции msgctl в разделе 6.5 и измените программу в листинге 3.2 так, чтобы выводился не только идентификатор, но и поле seq структуры ipc_perm.
2. Непосредственно после выполнения программы листинга 3.2 мы запускаем программу, создающую две очереди сообщений. Предполагая, что никакие другие приложения не использовали очереди сообщений с момента загрузки системы, определите, какие значения будут возвращены функцией msgget в качестве идентификаторов очередей сообщений.
3. В разделе 3.5 было отмечено, что функции getXXX System V IPC не используют маску создания файла. Напишите тестовую программу, создающую канал FIFO (с помощью функции mkfifo, описанной в разделе 4.6) и очередь сообщений System V, указав для обоих разрешение 666 (в восьмеричном формате). Сравните разрешения для созданных объектов (FIFO и очередь сообщений). Перед запуском программы удостоверьтесь, что значение umask отлично от нуля.
4. Серверу нужно создать уникальную очередь сообщений для своих клиентов. Что предпочтительнее: использовать какое-либо постоянное имя файла (например, имя сервера) в качестве аргумента функции ftok или использовать ключ IPC_PRIVATE?
5. Измените листинг 3.1 так, чтобы выводился только ключ IPC и путь к файлу. Запустите программу find, чтобы вывести список всех файлов вашей файловой системы, и передайте вывод вашей только что созданной программе. Скольким именам файлов будет соответствовать один и тот же ключ?
6. Если в вашей системе есть программа sar (system activity reporter — информация об активности системы), запустите команду

На экран будет выведено количество операций в секунду с очередями сообщений и семафорами, замеряемыми каждые 5 секунд 6 раз.

4.1. Введение

Неименованные каналы — это самая первая форма IPC в Unix, появившаяся еще в 1973 году в третьей версии (Third Edition [17]). Несмотря на полезность во многих случаях, главным недостатком неименованных каналов является отсутствие имени, вследствие чего они могут использоваться для взаимодействия только родственными процессами. Это было исправлено в Unix System III (1982) добавлением каналов FIFO, которые иногда называются именованными каналами. Доступ и к именованным каналам, и к неименованным организуется с помощью обычных функций `read` и `write`.

ПРИМЕЧАНИЕ

Программные (неименованные) каналы в принципе могут использоваться неродственными процессами, если предоставить им возможность передавать друг другу дескрипторы (см. раздел 15.8 этой книги или раздел 13.7 [24]). Однако на практике эти каналы обычно используются для осуществления взаимодействия между процессами, у которых есть общий предок.

В этой главе описываются детали, касающиеся создания и использования программных каналов и каналов FIFO. Мы рассмотрим пример простейшего сервера файлов, а также обратим внимание на некоторые детали модели клиент-сервер, в частности постараемся определить количество требуемых каналов IPC, сравним последовательные серверы с параллельными и неструктурированные потоки байтов с сообщениями.

4.2. Приложение типа клиент-сервер

Пример приложения модели клиент-сервер приведен на рис. 4.1. Именно на него мы будем ссылаться в тексте этой главы и главы 6 при необходимости проиллюстрировать использование программных каналов, FIFO и очередей сообщений System V.

Клиент считывает полное имя (файла) из стандартного потока ввода и записывает его в канал IPC. Сервер считывает это имя из канала IPC и производит попытку открытия файла на чтение. Если попытка оказывается успешной, сервер считывает файл и записывает его в канал IPC. В противном случае сервер возвращает клиенту сообщение об ошибке. Клиент считывает данные из канала IPC и записывает их в стандартный поток вывода. Если сервер не может считать файл, из канала будет считано сообщение об ошибке. В противном случае будет принято содержимое файла. Две штриховые линии между клиентом и сервером на рис. 4.1 представляют собой канал IPC.



Рис. 4.1. Пример приложения типа клиент-сервер

Программные каналы имеются во всех существующих реализациях и версиях Unix. Канал создается вызовом `pipe` и предоставляет возможность односторонней (односторонней) передачи данных:

Функция возвращает два файловых дескриптора: `fd[0]` и `fd[1]`, причем первый открыт для чтения, а второй — для записи.

ПРИМЕЧАНИЕ

Некоторые версии Unix, в частности SVR4, поддерживают двусторонние каналы (full-duplex pipes). В этом случае канал открыт на запись и чтение с обоих концов. Другой способ создания двустороннего канала IPC заключается в вызове функции `socketpair`, описанной в разделе 14.3 [24]. Его можно использовать в большинстве современных версий Unix. Однако чаще всего каналы используются при работе с интерпретатором команд, где уместно использование именно односторонних каналов.

Стандарты Posix.1 и Unix 98 требуют только односторонних каналов, и мы будем исходить из этого.

Для определения типа дескриптора (файла, программного канала или FIFO) можно использовать макрос `S_ISFIFO`. Он принимает единственный аргумент: поле `st_mode` структуры `stat` и возвращает значение «истина» (ненулевое значение) или «ложь» (ноль). Структуру `stat` для канала возвращает функция `fstat`. Для FIFO структура возвращается функциями `fstat`, `lstat` и `stat`.

На рис. 4.2 изображен канал при использовании его единственным процессом.



Рис. 4.2. Канал в одиночном процессе

Хотя канал создается одним процессом, он редко используется только этим процессом (пример канала в одиночном процессе приведен в листинге 5.12). Каналы обычно используются для связи между двумя процессами (родительским и дочерним) следующим образом: процесс создает канал, а затем вызывает `fork`, создавая свою копию — дочерний процесс (рис. 4.3). Затем родительский процесс закрывает открытый для чтения конец канала, а дочерний, в свою очередь, — открытый на запись конец канала. Это обеспечивает одностороннюю передачу данных между процессами, как показано на рис. 4.4.



Рис. 4.3. Канал после вызова `fork`



Рис. 4.4. Канал между двумя процессами

При вводе команды наподобие

в интерпретаторе команд Unix интерпретатор выполняет вышеописанные действия для создания трех процессов с двумя каналами между ними. Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись — к стандартному потоку вывода. Созданный таким образом канал изображен на рис. 4.5.



Рис. 4.5. Каналы между тремя процессами при конвейерной обработке

Все рассмотренные выше каналы были односторонними (односторонними), то есть позволяли передавать данные только в одну сторону. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону. Этапы создания двунаправленного канала IPC следующие:

1. Создаются каналы 1 (`fd1[0]` и `fd1[1]`) и 2 (`fd2[0]` и `fd2[1]`).
2. Вызов `fork`.
3. Родительский процесс закрывает доступный для чтения конец канала 1 (`fd1[0]`).

4. Родительский процесс закрывает доступный для записи конец канала 2 (fd2[1]).
5. Дочерний процесс закрывает доступный для записи конец канала 1 (fd1[1]).
6. Дочерний процесс закрывает доступный для чтения конец канала 2 (fd2[0]).

Текст программы, выполняющей эти действия, приведен в листинге 4.1. При этом создается структура каналов, изображенная на рис. 4.6.



Рис. 4.6. Двусторонняя передача данных по двум каналам

Пример

Давайте напишем программу, описанную в разделе 4.2, с использованием каналов. Функция `main` создает два канала и вызывает `fork` для создания копии процесса. Родительский процесс становится клиентом, а дочерний — сервером. Первый канал используется для передачи полного имени от клиента серверу, а второй — для передачи содержимого файла (или сообщения об ошибке) от сервера клиенту. Таким образом мы получаем структуру, изображенную на рис. 4.7.



Рис. 4.7. Реализация рис. 4.1 с использованием двух каналов

Обратите внимание на то, что мы изображаем на рис. 4.7 два канала, соединяющих сервер с клиентом, но оба канала проходят через ядро, поэтому каждый передаваемый байт пересекает интерфейс ядра дважды: при записи в канал и при считывании из него.

В листинге 4.1 приведена функция `main` для данного примера.

8-19 Создаются два канала и выполняются шесть шагов, уже упоминавшиеся в отношении рис. 4.6. Родительский процесс вызывает функцию `client` (листинг 4.2), а дочерний — функцию `server` (листинг 4.3).

20 Процесс-сервер (дочерний процесс) завершает свою работу первым, вызывая функцию `exit` после завершения записи данных в канал. После этого он становится процессом-зомби. Процессом-зомби называется дочерний процесс, завершивший свою работу, родитель которого еще функционирует, но не получил сигнал о завершении работы дочернего процесса. При завершении работы дочернего процесса ядро посыпает его родителю сигнал `SIGCHLD`, но родитель его не принимает и этот сигнал по умолчанию игнорируется. После этого функция `client` родительского процесса возвращает управление функции `main`, закончив Считывание данных из канала. Затем родительский процесс вызывает `waitpid` для получения информации о статусе дочернего процесса (зомби). Если родительский процесс не вызовет `waitpid`, а просто завершит работу, клиент будет унаследован процессом `init`, которому будет послан еще один сигнал `SIGCHLD`.

Функция `client` приведена в листинге 4.2.

8-14 Полное имя файла считывается из стандартного потока ввода и записывается в канал после удаления завершающего символа перевода строки, возвращаемого функцией `fgets`.

15-17 Затем клиент считывает все, что сервер направляет в канал, и записывает эти данные в стандартный поток вывода. Ожидается, что это будет содержимое файла, но в случае его отсутствия будет принято и записано в стандартный поток вывода сообщение об ошибке.

В листинге 4.3 приведена функция `server`.

8-11 Записанное в канал клиентом имя файла считывается сервером и дополняется завершающим символом с кодом 0 (null-terminated). Обратите внимание, что функция `read` возвращает данные, как только они помещаются в поток, не ожидая накопления некоторого их количества (`MAXLINE` в данном примере).

12-17 Файл открывается для чтения и при возникновении ошибки сообщение о ней возвращается клиенту с помощью канала. Для получения строки с соответствующим значению переменной `errno` сообщением об ошибке вызывается функция `strerror` (в книге [24, с. 690-691] вы найдете более подробный рассказ об этой функции).

18-23 При успешном завершении работы функции `open` содержимое файла копируется в канал.

Ниже приведен результат работы программы в случае наличия файла с указанным именем и в случае возникновения ошибок:

4.4. Двусторонние каналы

В предыдущем разделе мы отметили, что во многих системах реализованы двусторонние каналы. В Unix SVR4 это обеспечивается самой функцией `pipe`, а во многих других ядрах — функцией `socketpair`. Но что в действительности представляет собой двусторонний канал? Представим себе сначала односторонний канал, изображенный на рис. 4.8.



Рис. 4.8. Односторонний канал

Двусторонний канал мог бы быть реализован так, как это изображено на рис. 4.9. В этом случае неявно предполагается существование единственного буфера, в который помещается все, что записывается в канал (с любого конца, то есть дескриптора), и при чтении из канала данные просточитываются из буфера.



Рис. 4.9. Одна из возможных реализаций двустороннего канала (неправильная)

Такая реализация вызовет проблемы, например, в программе листинга A.14. Здесь требуется двусторонняя передача информации, причем потоки данных должны быть независимы. В противном случае некоторый процесс, записав данные в канал и перейдя затем в режим чтения из этого же канала, рискует считать обратно те же данные, которые были им только что туда записаны.

На рис. 4.10 изображена правильная реализация двустороннего канала.



Рис. 4.10. Правильная реализация двустороннего канала

Здесь двусторонний канал получается из объединения двух односторонних. Все данные, записываемые в `fd[1]`, будут доступны для чтения из `fd[0]`, а данные, записываемые в `fd[0]`, будут доступны для чтения из `fd[1]`.

Программа в листинге 4.4 иллюстрирует использование одного двустороннего канала для двусторонней передачи информации.

В этой программе сначала создается двусторонний канал, затем делается системный вызов `fork`. Породивший процесс записывает символ `r` в канал, а затем считывает из канала данные. Дочерний процесс ждет три секунды, считывает символ из канала, а потом записывает туда символ `c`. Задержка чтения для дочернего процесса позволяет породившему процессу вызвать `read` первым — таким образом мы можем узнать, не будет ли им считан обратно только что записанный символ.

При запуске этой программы в Solaris 2.6, в которой организована поддержка двусторонних каналов, мы получим ожидаемый результат:

Символ `r` передается по одному из двух односторонних каналов, изображенных на рис. 4.10, а именно по верхнему каналу. Символ `c` передается по нижнему одностороннему каналу. Родительский процесс не считывает обратно записанный им в канал символ `r` (что и требуется).

При запуске этой программы в Digital Unix 4.0B, в которой по умолчанию создаются односторонние каналы (двусторонние каналы — как в SVR4 — будут создаваться в том случае, если при компиляции указать специальные параметры), мы увидим результат, ожидаемый для одностороннего канала:

Родительский процесс записывает символ `r`, который успешно считывается дочерним процессом, однако при попытке считывания из канала (дескриптор `fd[1]`) родительский процесс прерывается с ошибкой, как и дочерний процесс, при попытке записи в канал (дескриптор `fd[0]`). Вспомните рис. 4.8. Функция `read` возвращает код ошибки `EBADF`, означающий, что дескриптор не открыт для чтения. Аналогично `write` возвращает тот же код ошибки, если дескриптор не был открыт на запись.

Другим примером использования каналов является имеющаяся в стандартной библиотеке ввода-вывода функция `ropen`, которая создает канал и запускает другой процесс, записывающий данные в этот канал или считывающий их из него:

Аргумент `command` представляет собой команду интерпретатора. Он обрабатывается программой `sh` (обычно это интерпретатор Bourne shell), поэтому для поиска исполняемого файла, вызываемого командой `command`, используется переменная `PATH`. Канал создается между вызывающим процессом и указанной командой. Возвращаемое функцией `ropen` значение представляет собой обычный указатель на тип `FILE`, который может использоваться для ввода или для вывода в зависимости от содержимого строки `type`:

- если `type` имеет значение `r`, вызывающий процесс считывает данные, направляемые командой `command` в стандартный поток вывода;
- если `type` имеет значение `w`, вызывающий процесс записывает данные в стандартный поток ввода команды `command`.

Функция `pclose` закрывает стандартный поток ввода-вывода `stream`, созданный командой `ropen`, ждет завершения работы программы и возвращает код завершения, принимаемый от интерпретатора.

ПРИМЕЧАНИЕ

Информацию о реализациях `ropen` и `pclose` можно найти в разделе 14.3 [21].

Пример

В листинге 4.5 изображено еще одно решение задачи с клиентом и сервером, использующее функцию popen и программу (утилиту Unix) cat.

8-17 Полное имя файла считывается из стандартного потока ввода, как и в программе в листинге 4.2. Формируется командная строка, которая передается popen. Вывод интерпретатора команд или команды cat копируется в стандартный поток вывода.

Одним из отличий этой реализации от приведенной в листинге 4.1 является отсутствие возможности формировать собственные сообщения об ошибках. Теперь мы целиком зависим от программы cat, а выводимые ею сообщения не всегда адекватны. Например, в системе Solaris 2.6 при попытке считать данные из файла, доступ на чтение к которому для нас запрещен, будет выведена следующая ошибка:

А в BSD/OS 3.1 мы получим более информативное сообщение в аналогичной ситуации:

Обратите также внимание на тот факт, что вызов popen в данном случае оказывается успешным, однако при первом же вызове fgets будет возвращен символ конца файла (EOF). Программа cat записывает сообщение об ошибке в стандартный поток сообщений об ошибках (stderr), а popen с этим потоком не связывается — к создаваемому каналу подключается только стандартный поток вывода.

Программные каналы не имеют имен, и их главным недостатком является невозможность передачи информации между неродственными процессами. Два неродственных процесса не могут создать канал для связи между собой (если не передавать дескриптор).

Аббревиатура FIFO расшифровывается как «first in, first out» — «первым вошел, первым вышел», то есть эти каналы работают как очереди. Именованные каналы в Unix функционируют подобно неименованным — они позволяют передавать данные только в одну сторону. Однако в отличие от программных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO.

FIFO создается функцией `mkfifo`:

Здесь *pathname* — обычное для Unix полное имя файла, которое и будет именем FIFO.

Аргумент mode указывает битовую маску разрешений доступа к файлу, аналогично второму аргументу команды `open`. В табл. 2.3 приведены шесть констант, определенных в заголовке `<sys/stat.h>`. Эти константы могут использоваться для задания разрешений доступа и к FIFO.

Функция `mkfifo` действует как `open`, вызванная с аргументом `O_CREAT | O_EXCL`. Это означает, что создается новый канал FIFO или возвращается ошибка `EEXIST`, в случае если канал с заданным полным именем уже существует. Если не требуется создавать новый канал, вызывайте `open` вместо `mkfifo`. Для открытия существующего канала или создания нового в том случае, если его еще не существует, вызовите `mkfifo`, проверьте, не возвращена ли ошибка `EEXIST`, и если такое случится, вызовите функцию `open`.

Команда `mkfifo` также создает канал FIFO. Ею можно пользоваться в сценариях интерпретатора или из командной строки.

После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции `open`, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, `fopen`). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись, поскольку именованные каналы могут быть только односторонними.

При записи в программный канал или канал FIFO вызовом `write` данные всегда добавляются к уже имеющимся, а вызов `read` считывает данные, помещенные в программный канал или FIFO первыми. При вызове функции `lseek` для программного канала или FIFO будет возвращена ошибка `ESPIPE`.

Пример

Переделаем программу, приведенную в листинге 4.1, таким образом, чтобы использовать два канала FIFO вместо двух программных каналов. Функции `client` и `server` останутся прежними; отличия появятся только в функции `main`, новый текст которой приведен в листинге 4.6.

10-16 В файловой системе в каталоге `/tmp` создается два канала. Если какой-либо из них уже существует — ничего страшного. Константа `FILE_MODE` определена в нашем заголовке `unprpc.h` (листинг B.1) как

При этом владельцу файла разрешается чтение и запись в него, а группе и прочим пользователям — только чтение. Эти биты разрешений накладываются на маску режима доступа создаваемых файлов (file mode creation mask) процесса.

17-27 Далее происходит вызов `fork`, дочерний процесс вызывает функцию `server` (листинг 4.3), а родительский процесс вызывает функцию `client` (листинг 4.2). Перед вызовом этих функций родительский процесс открывает первый канал на запись, а второй на чтение, в то время как дочерний процесс открывает первый канал на чтение, а второй — на запись. Картина аналогична примеру с каналами и иллюстрируется рис. 4.11.



Рис. 4.11. Приложение клиент-сервер, использующее две очереди

Изменения по сравнению с примером, в котором использовались программные каналы, следующие:

- Для создания и открытия программного канала требуется только один вызов — `pipe`. Для создания и открытия FIFO требуется вызов `mkfifo` и последующий вызов `open`.
- Программный канал автоматически исчезает после того, как будет закрыт последним использующим его процессом. Канал FIFO удаляется из файловой системы только после вызова `unlink`. Польза от лишнего вызова, необходимого для создания FIFO, следующая: канал FIFO получает имя в файловой системе, что позволяет одному процессу создать такой канал, а другому открыть его, даже если последний не является родственным первому. С программными каналами это неосуществимо.

В программах, некорректно использующих каналы FIFO, могут возникать неочевидные проблемы. Рассмотрим, например, листинг 4.6: если поменять порядок двух вызовов функции `open` в породившем процессе, программа перестанет работать. Причина в том, что чтение из FIFO блокирует процесс, если канал еще не открыт на запись каким-либо другим процессом. Действительно, если мы меняем порядок вызовов `open` в породившем процессе, и породивший, и порожденный процессы открывают канал на чтение, притом что на запись он еще не открыт, так что оба процесса блокируются. Такая ситуация называется блокированием, или зависанием (`deadlock`). Она будет рассмотрена подробно в следующем разделе.

Пример: неродственные клиент и сервер

В листинге 4.6 клиент и сервер все еще являлись родственными процессами. Переделаем этот пример так, чтобы родство между ними отсутствовало. В листинге 4.7 приведен текст программы-сервера. Текст практически идентичен той части программы из листинга 4.6, которая относилась к серверу.

Содержимое заголовка fifo.h приведено в листинге 4.8. Этот файл определяет имена двух FIFO, которые должны быть известны как клиенту, так и серверу.

В листинге 4.9 приведен текст программы-клиента, которая не слишком отличается от части программы из листинга 4.6, относящейся к клиенту. Обратите внимание, что именно клиент, а не сервер удаляет канал FIFO по завершении работы, потому что последние операции с этим каналом выполняются им.

ПРИМЕЧАНИЕ

Для программных каналов и каналов FIFO ядро ведет подсчет числа открытых дескрипторов, относящихся к ним, поэтому безразлично, кто именно вызовет unlink — клиент или сервер. Хотя эта функция и удаляет файл из файловой системы, она не влияет на открытые в момент ее выполнения дескрипторы. Однако для других форм IPC, таких как очереди сообщений стандарта System V, счетчик отсутствует, и если сервер удалит очередь после записи в нее последнего сообщения, она может быть удалена еще до того, как клиент это сообщение считает.

Для запуска клиента и сервера запустите сервер в фоновом режиме:

а затем запустите клиент. Можно было сделать и по-другому: запускать только программу-клиент, которая запускала бы сервер с помощью fork и exec. Клиент мог бы передавать серверу имена FIFO в качестве аргументов командной строки в команде exec, вместо того чтобы обе программы считывали их из заголовка. Но в этом случае сервер являлся бы дочерним процессом и проще было бы обойтись программным каналом.

4.7. Некоторые свойства именованных и неименованных каналов

Некоторые свойства именованных и неименованных каналов, относящиеся к их открытию, а также чтению и записи данных, заслуживают более пристального внимания. Прежде всего можно сделать дескриптор неблокируемым двумя способами.

1. При вызове `open` указать флаг `O_NONBLOCK`. Например, первый вызов `open` в листинге 4.9 мог бы выглядеть так:

2. Если дескриптор уже открыт, можно использовать `fcntl` для включения флага `O_NONBLOCK`. Этот прием нужно применять для программных каналов, поскольку для них не вызывается функция `open` и нет возможности указать флаг `O_NONBLOCK` при ее вызове. Используя `fcntl`, мы сначала получаем текущий статус файла с помощью `F_GETFL`, затем добавляем к нему с помощью побитового логического сложения (OR) флаг `O_NONBLOCK` и записываем новый статус с помощью команды `F_SETFL`:

Будьте аккуратны с программами, которые просто устанавливают требуемый флаг, поскольку при этом сбрасываются все прочие флаги состояния:

Таблица 4.1 иллюстрирует действие флага, отключающего блокировку, при открытии очереди и при чтении данных из пустого программного канала или канала FIFO.

Таблица 4.1. Действие флага `O_NONBLOCK` на именованные и неименованные каналы

Запомните несколько дополнительных правил, действующих при чтении и записи данных в программные каналы и FIFO.

- При попытке считать больше данных, чем в данный момент содержится в программном канале или FIFO, возвращается только имеющийся объем данных. Нужно предусмотреть обработку ситуации, в которой функция `read` возвращает меньше данных, чем было запрошено.
- Если количество байтов, направленных на запись функции `write`, не превышает значения `PIPE_BUF` (ограничение, устанавливаемое стандартом Posix, о котором более подробно рассказывается в разделе 4.11), то ядро гарантирует *атомарность* операции записи. Это означает, что если два процесса запишут данные в программный канал или FIFO приблизительно одновременно, то в буфер будут помещены сначала все данные от первого процесса, а затем от второго, либо наоборот. Данные от двух процессов при этом не будут смешиваться. Однако если количество байтов превышает значение `PIPEBUF`, атомарность операции записи не гарантируется.

ПРИМЕЧАНИЕ

Posix.1 требует, чтобы значение `PIPE_BUF` равнялось по меньшей мере 512. Характерные значения, встречающиеся на практике, лежат в диапазоне от 1024 (BSD/OS 3.1) до 5120 байт (Solaris 2.6). В разделе 4.11 приведен текст программы, выводящей значение этой константы.

■ Установка флага `O_NONBLOCK` не влияет на атомарность операции записи в программный канал или FIFO — она определяется исключительно объемом посылаемых данных в сравнении с величиной `PIPE_BUF`. Однако если для программного канала или FIFO отключена блокировка, возвращаемое функцией `write` значение зависит от количества байтов, отправленных на запись, и наличия свободного места в программном канале или FIFO. Если количество байтов не превышает величины `PIPE_BUF`, то:

- Если в канале достаточно места для записи требуемого количества данных, они будут переданы все сразу.
 - Если места в программном канале или FIFO недостаточно для записи требуемого объема данных, происходит немедленное завершение работы функции с возвратом ошибки `EAGAIN`. Поскольку установлен флаг `O_NONBLOCK`, процесс не может быть заблокирован, но в то же время ядро не может принять лишь часть данных, так как при этом невозможно гарантировать атомарность операции записи. Поэтому ядро возвращает ошибку, сообщающую процессу о необходимости попытаться произвести запись еще раз.
- Если количество байтов превышает значение `PIPE_BUF`, то:
- Если в программном канале или FIFO есть место хотя бы для одного байта, ядро передает в буфер ровно столько данных, сколько туда может поместиться, и это переданное количество возвращается функцией `write`.
 - Если в программном канале или FIFO свободное место отсутствует, происходит немедленное

завершение работы с возвратом ошибки EAGAIN.

■ При записи в программный канал или FIFO, не открытый для чтения, ядро посыпает сигнал SIGPIPE:

□ Если процесс не принимает (catch) и не игнорирует SIGPIPE, выполняется действие по умолчанию — завершение работы процесса.

□ Если процесс игнорирует сигнал SIGPIPE или перехватывает его и возвращается из подпрограммы его обработки, write возвращает ошибку с кодом EPIPE.

ПРИМЕЧАНИЕ

SIGPIPE считается синхронным сигналом, что означает, что он привязан к конкретному программному потоку, а именно тому, который вызвал функцию write. Простейшим способом обработки сигнала является его игнорирование (установка SIG_IGN) и предоставление функции write возможности вернуть ошибку с кодом EPIPE. В приложении всегда должна быть предусмотрена обработка ошибок, возвращаемых функцией write, а вот определить, что процесс был завершен сигналом SIGPIPE, сложнее. Если сигнал не перехватывается, придется посмотреть на статус завершения работы процесса (termination status) из интерпретатора команд, чтобы узнать, что процесс был принудительно завершен сигналом и каким именно сигналом. В разделе 5.13 [24] о сигнале SIGPIPE рассказывается более подробно.

Преимущества канала FIFO проявляются более явно в том случае, когда сервер представляет собой некоторый длительно функционирующий процесс (например, демон, наподобие описанного в главе 12 [24]), не являющийся родственным клиенту. Демон создает именованный канал с вполне определенным известным именем, открывает его на чтение, а запускаемые впоследствии клиенты открывают его на запись и отправляют демону команды и необходимые данные. Односторонняя связь в этом направлении (от клиента к серверу) легко реализуется с помощью FIFO, однако необходимость отправки данных в обратную сторону (от сервера к клиенту) усложняет задачу.

Рисунок 4.12 иллюстрирует прием, применяемый в этом случае.



Рис. 4.12. Один сервер, несколько клиентов

Сервер создает канал с известным полным именем, в данном случае /tmp/fifo.serv. Из этого канала он считывает запросы клиентов. Каждый клиент при запуске создает свой собственный канал, полное имя которого определяется его идентификатором процесса. Клиент отправляет свой запрос в канал сервера с известным именем, причем запрос этот содержит идентификатор процесса клиента и имя файла, отправку которого клиент запрашивает у сервера. В листинге 4.10 приведен текст программы сервера.

10-15 Сервер создает канал FIFO с известным именем, обрабатывая ситуацию, когда такой канал уже существует. Затем этот канал открывается дважды: один раз только для чтения, а второй — только для записи. Дескриптор readfifo используется для приема запросов от клиентов, а дескриптор dummyfd не используется вовсе. Причина, по которой нужно открыть канал для записи, видна из табл. 4.1. Если канал не открыт на запись, то при завершении работы очередного клиента этот канал будет опустошаться и сервер будет считывать 0, означающий конец файла. Пришлось бы каждый раз закрывать канал вызовом close, а затем заново открывать его с флагом O_RDONLY, что приводило бы к блокированию демона до подключения следующего клиента. Мы же всегда будем иметь дескриптор, открытый на запись, поэтому функция read не будет возвращать 0, означающий конец файла, при отсутствии клиентов. Вместо этого сервер просто будет блокироваться при вызове read, ожидая подключения следующего клиента. Этот трюк упрощает код программы-сервера и уменьшает количество вызовов open для канала сервера.

При запуске сервера первый вызов open (с флагом O_RDONLY) приводит к блокированию процесса до появления первого клиента, открывающего канал сервера на запись (см. табл. 4.1). Второй вызов open (с флагом O_WRONLY) не приводит к блокированию, поскольку канал уже открыт на запись.

16 Каждый запрос, принимаемый от клиента, представляет собой одну строку, состоящую из идентификатора процесса, пробела и полного имени требуемого файла. Эта строка считывается функцией readline (приведенной в [24, с. 79]).

17-26 Символ перевода строки, возвращаемый функцией readline, удаляется. Этот символ может отсутствовать только в том случае, если буфер был заполнен, прежде чем был обнаружен символ перевода строки, либо если последняя введенная строка не была завершена этим символом. Функция strchr возвращает указатель на первый пробел в этой строке, который затем увеличивается на единицу, чтобы он указывал на первый символ полного имени файла, следующего за пробелом. Полное имя канала клиента формируется из его идентификатора процесса, и этот канал открывается сервером на запись.

Открытие файла и отправка его в FIFO клиента

27-44 Оставшаяся часть кода программы-сервера аналогична функции server из листинга 4.3. Программа открывает файл; если при этом возникает ошибка — клиенту отсылается сообщение о ней. Если открытие файла завершается успешно, его содержимое копируется в канал клиента. После завершения копирования открытый сервером «конец» (дескриптор) канала клиента должен быть закрыт с помощью функции close, чтобы функция read вернула программе-клиенту значение 0 (конец файла). Сервер не удаляет канал клиента; клиент должен самостоятельно позаботиться об этом после приема от сервера символа конца файла. Текст программы-клиента приведен в листинге 4.11.

10-14 Идентификатор процесса клиента содержится в имени создаваемого им канала.

15-21 Запрос клиента состоит из его идентификатора процесса, одного пробела, полного имени запрашиваемого им файла и символа перевода строки. Стока запроса формируется в массиве buff, причем имя файла считывается из стандартного потока ввода.

22-24 Клиент открывает канал сервера и записывает в него строку запроса. Если клиент окажется первым с момента запуска сервера, вызов open разблокирует сервер, заблокированный после сделанного им вызова open (с флагом O_RDONLY).

25-31 Ответ сервера считывается из канала и записывается в стандартный поток вывода, после чего канал клиента закрывается и* удаляется.

Сервер может быть запущен в одном из окон, а клиент — в другом, и программа будет работать так, как мы и рассчитывали. Ниже мы приводим только текст, выводимый клиентом:

Мы можем также связаться с сервером из интерпретатора команд, поскольку каналы FIFO обладают именами в файловой системе.

Мы отсылаем серверу идентификатор процесса текущей копии интерпретатора и полное имя файла одной командой интерпретатора (echo) и считываем из канала сервера результат с помощью другой команды (cat). Между выполнением этих двух команд может пройти произвольный промежуток времени. Таким образом, сервер помещает содержимое файла в канал, а клиент затем запускает команду cat, чтобы считать оттуда данные. Может показаться, что данные каким-то образом хранятся в канале, хотя он не открыт ни одним процессом. На самом деле все не так. После закрытия программного канала или FIFO последним процессом с помощью команды close все данные, в нем находящиеся, теряются. В нашем примере сервер, считав строку запроса от клиента, блокируется при попытке открыть канал клиента, потому что клиент (наша копия интерпретатора) еще не открыл его на чтение (вспомним табл. 4.1). Только после вызова cat некоторое время спустя канал будет открыт на чтение, и тогда сервер разблокируется. Кстати, таким образом осуществляется атака типа «отказ в обслуживании» (denial-of-service attack), которую мы обсудим в следующем разделе.

Использование интерпретатора позволяет провести простейшую проверку способности сервера обрабатывать ошибки. Мы можем отправить серверу строку без идентификатора процесса или отослать ему такой идентификатор, которому не соответствует никакой канал FIFO в каталоге /tmp. Например, если мы запустим сервер и введем нижеследующие строки:

то сервер выдаст текст:

Атомарность записи в FIFO

Наша простейшая пара клиент-сервер позволяет наглядно показать важность наличия свойства атомарности записи в программные каналы и FIFO. Предположим, что два клиента посылают серверу запрос приблизительно в один и тот же момент. Первый клиент отправляет следующую строку:

второй:

Предполагая, что каждый клиент помещает данные в FIFO за один вызов `write` и каждая строка имеет размер, не превышающий величины `PIPE_BUF` (что чаще всего заведомо выполняется, поскольку `PIPE_BUF` обычно лежит в диапазоне 1024-5120, а длина полного имени обычно ограничена 1024 байт), мы можем гарантировать, что в FIFO данные будут иметь следующий вид:

либо

Данные в канале не могут смешаться в «кашу», наподобие:

FIFO и NFS

Каналы FIFO представляют собой вид IPC, который может использоваться только в пределах одного узла. Хотя FIFO и обладают именами в файловой системе, они могут применяться только в локальных файловых системах, но не в присоединенных сетевых (NFS).

В этом примере файловая система `/nfs/bsdi/usr` — это файловая система `/usr` на узле `bsdi`.

Некоторые системы (например, BSD/OS) позволяют создавать FIFO в присоединенных файловых системах, но по ним нельзя передавать данные между узлами. В этом случае такой канал может использоваться лишь как «точка рандеву» в файловой системе между клиентами и серверами на одном и том же узле. Процесс, выполняемый на одном узле, *не может* послать данные через FIFO процессу, выполняемому на другом узле, даже если оба процесса смогут открыть этот канал, доступный обоим узлам через сетевую файловую систему.

Сервер в нашем простом примере из предыдущего раздела являлся последовательным сервером (iterative server). Он последовательно обрабатывал запросы клиентов, переходя к следующему только после полного завершения работы с предыдущим. Например, если два клиента пошлют запрос такому серверу приблизительно одновременно, причем один из них запросит 10-мегабайтный файл, отправка которого займет, например, 10 секунд, а второй — 10-байтный файл, то второму придется ждать по меньшей мере 10 секунд, пока не будет обслужен первый клиент.

Альтернативой является параллельный сервер (concurrent server). Наиболее часто встречаемый в Unix вид такого сервера называется one-child-per-client (каждому клиенту — один дочерний процесс). Сервер вызывает fork для создания нового процесса каждый раз, когда появляется новый клиент. Дочерний процесс полностью обрабатывает запрос клиента, а поддержка многозадачности в Unix обеспечивает параллельность выполнения всех этих процессов. Однако существуют и другие методы решения задачи, подробно описанные в главе 27 [24]:

- создание пула дочерних процессов и передача нового клиента свободному дочернему процессу;
- создание одного программного потока для каждого клиента;
- создание пула потоков и передача нового клиента свободному потоку.

Хотя в [24] обсуждаются проблемы создания сетевых серверов, те же методы применимы и к серверам межпроцессного взаимодействия (IPC server), клиенты которых находятся на одном узле.

Атака типа «отказ в обслуживании»

Один из недостатков последовательных серверов был уже отмечен выше — некоторым клиентам приходится ждать дольше чем нужно, потому что их запросы приходят после запросов других клиентов, запрашивающих большие файлы. Существует и другая проблема. Вспомним наш пример с интерпретатором команд, приведенный после листинга 4.11, и относящееся к нему обсуждение того, что сервер блокируется при вызове open для FIFO клиента, если клиент еще не открыл этот канал (чего не происходит до выполнения cat). Это дает возможность злоумышленнику «подвесить» сервер, послав ему запрос, не открывая канала. Этот тип атаки называется «отказ в обслуживании» (Denial of Service — DoS). Чтобы исключить возможность такой атаки, нужно быть аккуратным при написании последовательной части любого сервера, учитывая возможность и потенциальную продолжительность его блокирования. Одним из методов решения проблемы является установка максимального времени ожидания для некоторых операций, однако обычно проще сделать сервер параллельным, а не последовательным, поскольку в данном случае атака будет действовать лишь на один из дочерних процессов, а не на весь сервер. Однако даже параллельный сервер не защищен от атаки полностью: злоумышленник все еще может послать множество запросов, что приведет к превышению предела количества порожденных сервером процессов и невозможности выполнения последующих вызовов fork.

4.10. Потоки и сообщения

Приведенные примеры программных каналов и каналов FIFO использовали потоковую модель ввода-вывода, что естественно для Unix. При этом отсутствуют границы записей — данные при операциях чтения и записи не проверяются вовсе. Процесс, считывающий 100 байт из FIFO, не может определить, записал ли другой процесс в FIFO все 100 байт за 1 раз, или за 5 раз по 20 байт, или в любой другой комбинации общим объемом 100 байт. Возможно, один процесс записал в FIFO 55 байт, а потом другой — 45. Данные представляют собой просто поток байтов, никак не интерпретируемых системой. Если же требуется какая-либо интерпретация данных, пишущий и читающий процессы должны заранее «договориться» о ее правилах и выполнять всю работу самостоятельно.

Иногда приложению может потребоваться передавать данные, обладающие некоторой внутренней структурой. Это могут быть, например, сообщения переменной длины: в этом случае читающий процесс должен знать, где заканчивается одно сообщение и начинается следующее. Для разграничения сообщений широко используются три метода:

1. Специальная внутриполосная завершающая последовательность: множество приложений под Unix используют в качестве разделителя сообщений символ перевода строки. Пищий процесс добавляет к каждому сообщению этот символ, а считывающий процесс производит построчное считывание. Так работают клиент и сервер из листингов 4.10 и 4.11, чтобы разделить запросы клиентов. Этот метод требует исключения символа-разделителя из самих передаваемых данных (в случае необходимости его передать он должен предваряться другим специальным символом).
2. Явное указание длины: каждой записи предшествует информация об ее длине. Мы вскоре воспользуемся этим методом. Он также применяется в Sun RPC при использовании совместно с TCP. Одним из преимуществ этого метода является отсутствие необходимости исключать разделитель из передаваемых данных, поскольку получатель не проверяет все данные, а переходит сразу к концу очередной записи, чтобы узнать длину следующей.
3. Одна запись за подключение: приложение закрывает подключение к партнеру (подключение TCP для сетевых приложений либо просто подключение IPC), обозначая конец записи. Это требует повторного подключения для передачи следующей записи, однако используется в стандарте HTTP 1.0.

Стандартная библиотека ввода-вывода также может использоваться для считывания и записи данных в программный канал или FIFO. Поскольку канал может быть открыт только функцией `fopen`, возвращающей открытый дескриптор, для создания нового стандартного потока, связанного с этим дескриптором, можно использовать стандартную функцию `fdopen`. Канал FIFO обладает именем, поэтому он может быть открыт с помощью функции `fopen`.

Можно создавать и более структурированные сообщения — эта возможность предоставляется очередями сообщений и в Posix, и в System V. Мы вскоре узнаем, что каждое сообщение обладает длиной и приоритетом (типом в System V). Длина и приоритет указываются отправителем и возвращаются получателю после считывания сообщения. Каждое сообщение представляет собой запись, аналогично дейтаграммам UDP ([24]).

Мы можем структурировать данные, передаваемые по программному каналу или FIFO, самостоятельно. Определим сообщение в нашем заголовочном файле `mesg.h`, как показано в листинге 4.12.

Каждое сообщение содержит в себе информацию о своем типе (`mesg_type`), причем значение этой переменной должно быть больше нуля. Пока мы будем игнорировать это поле в записи, но вернемся к нему в главе 6, где описываются очереди сообщений System V. Каждое сообщение также обладает длиной, которая может быть и нулевой. Структура `tumesg` позволяет предварить каждое сообщение информацией о его типе и длине вместо использования символа перевода строки для сигнализации конца сообщения. Ранее мы отметили два преимущества этого подхода: получатель не должен сканировать все принятые байты в поисках конца сообщения и отсутствует необходимость исключать появление разделителя в самих данных.

На рис. 4.13 изображен вид структуры `tumesg` и ее использование с каналами, FIFO и очередями сообщений System V.



Рис. 4.13. Структура `tumesg`

Мы определяем две функции для отправки и приема сообщений. В листинге 4.13 приведен текст

функции `mesg_send`, а в листинге 4.14 — функции `mesg_recv`.

Теперь для каждого сообщения функция `read` вызывается дважды: один раз для считывания длины, а другой — для считывания самого сообщения (если его длина больше 0).

ПРИМЕЧАНИЕ

Внимательные читатели могли заметить, что функция `mesg_recv` проверяет наличие всех возможных ошибок и прекращает работу при их обнаружении. Однако мы все же определили функцию-обертку `Mesg_recv` и вызываем из наших программ именно ее — для единобразия.

Изменим теперь функции `client` и `server`, чтобы воспользоваться новыми функциями `mesg_send` и `mesg_recv`. В листинге 4.15 приведен текст функции-клиента.

8-16 Полное имя считывается из стандартного потока ввода и затем отправляется на сервер с помощью функции `mesg_send`.

17-19 Клиент вызывает функцию `mesg_recv` в цикле, считывая все приходящие от сервера сообщения. По соглашению, когда `mesg_recv` возвращает нулевую длину сообщения, это означает конец передаваемых сервером данных. Мы увидим, что сервер добавляет символ перевода строки к каждому сообщению, отправляемому клиенту, поэтому пустая строка будет иметь длину сообщения 1. В листинге 4.16 приведен текст функции-сервера.

8-18 Сервер принимает от клиента имя файла. Хотя значение `mesg_type`, равное 1, нигде не используется (оно затирается функцией `mesg_recv` из листинга 4.14), мы будем использовать ту же функцию при работе с очередями сообщений System V (листинг 6.8), а в данном случае в этом значении уже возникает потребность (см., например, листинг 6.11). Стандартная функция ввода-вывода `fopen` открывает файл, что отличается от листинга 4.3, где вызывалась функция `open` для получения дескриптора файла. Причина, по которой мы воспользовались `fopen`, заключается в том, что в этой программе мы пользуемся библиотечной функцией `fgets` для считывания содержимого файла построчно и затем отправляем клиенту строку за строкой.

19-26 Если вызов `fopen` оказывается успешным, содержимое файла считывается с помощью функции `fgets` и затем отправляется клиенту построчно. Сообщение с нулевой длиной означает конец файла.

При использовании программных каналов или FIFO мы могли бы также закрыть канал IPC, чтобы дать клиенту знать о том, что передача файла завершена. Однако мы используем передачу сообщения нулевой длины, потому что другие типы IPC не поддерживают концепцию конца файла.

Функции `main`, вызывающие новые функции `client` и `server`, вообще не претерпели никаких изменений. Мы можем использовать либо версию для работы с каналами (листинг 4.1), либо версию для работы с FIFO (листинг 4.6).

4.11. Ограничения программных каналов и FIFO

На программные каналы и каналы FIFO системой накладываются всего два ограничения:

- OPEN_MAX — максимальное количество дескрипторов, которые могут быть одновременно открыты некоторым процессом (Posix устанавливает для этой величины ограничение снизу — 16);
- PIPE_BUF — максимальное количество данных, для которого гарантируется атомарность операции записи (описано в разделе 4.7; Posix требует по меньшей мере 512 байт).

Значение OPEN_MAX можно узнать, вызвав функцию sysconf, как мы вскоре покажем. Обычно его можно изменить из интерпретатора команд с помощью команды ulimit (в Bourne shell и KornShell, как мы вскоре покажем) или с помощью команды limit (в C shell). Оно может быть изменено и самим процессом с помощью вызова функции setrlimit (подробно описана в разделе 7.11 [21]).

Значение PIPE_BUF обычно определено в заголовочном файле `<limits.h>`, но с точки зрения стандарта Posix оно представляет собой переменную, зависимую от полного имени файла. Это означает, что ее значение может меняться в зависимости от указываемого имени файла (для FIFO, поскольку каналы имен не имеют), поскольку разные имена могут относиться к разным файловым системам и эти файловые системы могут иметь различные характеристики. Это значение можно получить в момент выполнения программы, вызвав либо pathconf, либо fpathconf. В листинге 4.17 приведен пример, выводящий текущее значение этих двух ограничений.

Вот несколько примеров, в которых указываются имена файлов, относящиеся к различным файловым системам:

Покажем теперь, как изменить значение OPEN_MAX в Solaris, используя интерпретатор KornShell:

ПРИМЕЧАНИЕ

Хотя значение PIPE_BUF для FIFO, в принципе, может меняться в зависимости от файловой системы, к которой относится файл, на самом деле это очень редкий случай.

В главе 2 [21] описаны функции fpathconf, pathconf и sysconf, которые предоставляют информацию о некоторых ограничениях ядра во время выполнения программы. Стандарт Posix.1 определяет 12 констант, начинающихся с `_PC_`, и 52, начинающихся с `_SC_`. Системы Digital Unix 4.0B и Solaris 2.6 расширяют последнее ограничение, определяя около 100 констант, значения которых могут быть получены в момент выполнения программы с помощью sysconf.

Команда getconf определяется стандартом Posix.2 и выводит значения большинства этих ограничений. Например:

4.12. Резюме

Именованные и неименованные каналы представляют собой базовые строительные блоки для множества приложений. Программные каналы (неименованные) обычно используются в интерпретаторе команд, а также внутри программ — часто для передачи информации от дочернего процесса к родительскому. Можно исключить часть кода, относящегося к использованию каналов (`pipe`, `fork`, `close`, `exec` и `waitpid`), используя функции `ropen` и `pclose`, которые берут на себя все тонкости и запускают интерпретатор команд.

Каналы FIFO похожи на программные каналы, но создаются вызовом `mkfifo` и затем могут быть открыты с помощью функции `open`. При открытии FIFO следует быть аккуратным, поскольку процесс может быть заблокирован, а зависит это от множества условий (см. табл. 4.1).

Используя программные каналы и FIFO, мы создали несколько вариантов приложений типа клиент-сервер: один сервер с несколькими клиентами, последовательный и параллельный серверы. Последовательный сервер единовременно обрабатывает запрос только от одного клиента; такие серверы обычно уязвимы для атак типа «отказ в обслуживании». Параллельный сервер запускает отдельный процесс или поток для обработки запроса нового клиента.

Одним из свойств программных каналов и FIFO является то, что данные по ним передаются в виде потоков байтов, аналогично соединению TCP. Деление этого потока на самостоятельные записи целиком предоставляется приложению. Мы увидим в следующих двух главах, что очереди сообщений автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP.

Упражнения

1. При переходе от рис. 4.3 к рис. 4.4: что могло бы произойти, если бы дочерний процесс не закрывал дескриптор (`close(fd[1])`)?
2. Описывая `mkfifo` в разделе 4.6, мы сказали, что для открытия существующего FIFO или создания нового, если его не существует, следует вызвать `mkfifo`, проверить, не возвращается ли ошибка `EEXIST`, и вызвать `open`, если это происходит. Что если изменить логику и вызвать сначала `open`, а затем `mkfifo`, если FIFO не существует?
3. Что происходит при вызове `ropen` в листинге 4.5, если в интерпретаторе возникает ошибка?
4. Удалите вызов `open` для FIFO сервера в листинге 4.10 и проверьте, приведет ли это к завершению работы сервера после отключения последнего клиента.
5. К листингу 4.10: мы отметили, что при запуске сервера его работа блокируется при вызове первой функции `open`, пока FIFO не будет открыт на запись первым клиентом. Как можно обойти это таким образом, чтобы обе функции `open` завершали работу немедленно, а блокирование происходило при первом вызове `readline`?
6. Что произойдет с клиентом в листинге 4.11, если поменять порядок вызовов `open`?
7. Почему сигнал отправляется процессу, в котором канал FIFO открыт на запись, после отключения последнего читающего клиента, а не читающему клиенту после отключения последнего пишущего?
8. Напишите небольшую тестирующую программу для определения того, возвращает ли `fstat` количество байтов в FIFO в качестве поля `st_size` структуры `stat`.
9. Напишите небольшую тестирующую программу для определения того, что возвращает функция `select` при проверке возможности записи в дескриптор канала, у которого закрыт второй конец.

5.1. Введение

Очередь сообщений можно рассматривать как связный список сообщений. Программные потоки с соответствующими разрешениями могут помещать сообщения в очередь, а потоки с другими соответствующими разрешениями могут извлекать их оттуда. Каждое сообщение представляет собой запись (вспомните сравнение потоков и сообщений в разделе 4.10), и каждому сообщению его отправителем присваивается приоритет. Для записи сообщения в очередь не требуется наличия ожидающего его процесса. Это отличает очереди сообщений от программных каналов и FIFO, в которые нельзя произвести запись, пока не появится считывающий данные процесс.

Процесс может записать в очередь какие-то сообщения, после чего они могут быть получены другим процессом в любое время, даже если первый завершит свою работу. Мы говорим, что очереди сообщений обладают живучестью ядра (*kernel persistence*, раздел 1.3). Это также отличает их от программных каналов и FIFO. В главе 4 говорится о том, что данные, остающиеся в именованном или неименованном канале, сбрасываются, после того как все процессы закроют его.

В этой главе рассматриваются очереди сообщений стандарта Posix, а в главе 6 — стандарта System V. Функции для работы с ними во многом схожи, а главные отличия заключаются в следующем:

- операция считывания из очереди сообщений Posix всегда возвращает самое старое сообщение с наивысшим приоритетом, тогда как из очереди System V можно считать сообщение с произвольно указанным приоритетом;
- очереди сообщений Posix позволяют отправить сигнал или запустить программный поток при помещении сообщения в пустую очередь, тогда как для очередей System V ничего подобного не предусматривается.

Каждое сообщение в очереди состоит из следующих частей:

- приоритет (беззнаковое целое, Posix) либо тип сообщения (целое типа long, System V);
- длина полезной части сообщения, которая может быть нулевой;
- собственно данные (если длина сообщения отлична от 0).

Этим очереди сообщений отличаются от программных каналов и FIFO. Последние две части сообщения представляют собой байтовые потоки, в которых отсутствуют границы между сообщениями и никак не указывается их тип. Мы обсуждали этот вопрос в разделе 4.10 и добавили свой собственный интерфейс для пересылки сообщений по программным каналам и FIFO. На рис. 5.1 показан возможный вид очереди сообщений.



Рис. 5.1. Очередь сообщений Posix, содержащая три сообщения

Мы предполагаем реализацию через связный список, причем его заголовок содержит два атрибута очереди: максимально допустимое количество сообщений в ней и максимальный размер сообщения. Об этих атрибутах мы расскажем более подробно в разделе 5.3.

В этой главе мы используем метод, к которому будем прибегать и в дальнейшем, рассматривая очереди сообщений, семафоры и разделяемую память. Поскольку все эти объекты IPC обладают по крайней мере живучестью ядра (вспомните раздел 1.3), мы можем писать небольшие программы, использующие эти методы для экспериментирования с ними и получения большей информации о том, как они работают. Например, мы можем написать программу, создающую очередь сообщений Posix, а потом написать другую программу, которая помещает сообщение в такую очередь, а потом еще одну, которая будет считывать сообщения из очереди. Помещая в очередь сообщения с различным приоритетом, мы увидим, в каком порядке они будут возвращаться функцией `mq_receive`.

Функция `mq_open` создает новую очередь сообщений либо открывает существующую:

Требования к аргументу *name* описаны в разделе 2.2.

Аргумент *oflag* может принимать одно из следующих значений: `O_RDONLY`, `O_WRONLY`, `O_RDWR` в сочетании (логическое сложение) с `O_CREAT`, `O_EXCL`, `O_NONBLOCK`. Все эти флаги описаны в разделе 2.3.

При создании новой очереди (указан флаг `O_CREAT` и очередь сообщений еще не существует) требуется указание аргументов *mode* и *attr*. Возможные значения аргумента *mode* приведены в табл. 2.3. Аргумент *attr* позволяет задать некоторые атрибуты очереди. Если в качестве этого аргумента задать нулевой указатель, очередь будет создана с атрибутами по умолчанию. Эти атрибуты описаны в разделе 5.3.

Возвращаемое функцией `mq_open` значение называется дескриптором очереди сообщений, но оно не обязательно должно быть (и, скорее всего, не является) небольшим целым числом, как дескриптор файла или программного сокета. Это значение используется в качестве первого аргумента оставшихся семи функций для работы с очередями сообщений.

ПРИМЕЧАНИЕ

В системе Solaris 2.6 тип `mqd_t` определен как `void*`, а в Digital Unix 4.0B — как `int`. В нашем примере в разделе 5.8 эти дескрипторы трактуются как указатели на структуру. Название «дескриптор» было дано им по ошибке.

Открытая очередь сообщений закрывается функцией `mq_close`:

По действию эта функция аналогична `close` для открытого файла: вызвавший функцию процесс больше не может использовать дескриптор, но очередь сообщений из системы не удаляется. При завершении процесса все открытые очереди сообщений закрываются, как если бы для каждой был сделан вызов `mq_close`.

Для удаления из системы имени (*name*), которое использовалось в качестве аргумента при вызове `mq_open`, нужно использовать функцию `mq_unlink`:

Для очереди сообщений (как и для файла) ведется подсчет числа процессов, в которых она открыта в данный момент, и по действию эта функция аналогична `unlink` для файла: имя (*name*) может быть удалено из системы, даже пока число подключений к очереди отлично от нуля, но удаление очереди (в отличие от удаления имени из системы) не будет осуществлено до того, как очередь будет закрыта последним использовавшим ее процессом.

Очереди сообщений Posix обладают по меньшей мере живучестью ядра (раздел 1.3), то есть они продолжают существовать, храня все имеющиеся в них сообщения, даже если нет процессов, в которых они были бы открыты. Очередь существует, пока она не будет удалена явно с помощью `mq_unlink`.

ПРИМЕЧАНИЕ

Мы увидим, что если очередь сообщений реализована через отображаемые в память файлы (раздел 12.2), она может обладать живучестью файловой системы, но это не является обязательным и рассчитывать на это нельзя.

Пример: программа mqcreate1

Поскольку очереди сообщений Posix обладают по крайней мере живучестью ядра, можно написать набор небольших программ для работы с ними — с этими программами будет проще экспериментировать. Программа из листинга 5.1 создает очередь сообщений, имя которой принимается в качестве аргумента командной строки.

В командной строке можно указать параметр `-e`, управляющий исключающим созданием очереди. (О функции getopt и нашей обертке Getopt рассказано более подробно в комментарии к листингу 5.5.) При возвращении функция getopt сохраняет в переменной optind индекс следующего аргумента, подлежащего обработке.

Мы вызываем функцию `mq_open`, указывая ей в качестве имени IPC полученный из командной строки параметр, не обращаясь к рассмотренной нами в разделе 2.2 функции `px_ipc_name`. Это даст нам возможность узнать, как в данной реализации обрабатываются имена Posix IPC (мы используем для этого наши маленькие тестовые программы на протяжении всей книги).

Ниже приведен результат работы программы в Solaris 2.6:

Мы назвали эту версию программы `mqcreate1`, поскольку она будет улучшена в листинге 5.4, после того как мы обсудим использование атрибутов очереди. Разрешения на доступ к третьему файлу определяются константой `FILE_MODE` (чтение и запись для пользователя, только чтение для группы и прочих пользователей), но у двух первых файлов разрешения отличаются. Можно предположить, что в файле с буквой D в имени хранятся данные; файл с буквой L представляет собой какую-то блокировку, а в файле с буквой R хранятся разрешения.

В Digital Unix 4.0B мы указываем действительное имя создаваемого файла:

Пример: программа mqunlink

В листинге 5.2 приведена программа mqunlink, удаляющая из системы очередь сообщений.

С помощью этой программы мы можем удалить очередь сообщений, созданную программой mqcreate1:

При этом будут удалены все три файла из каталога /tmp, которые относятся к этой очереди.

У каждой очереди сообщений имеются четыре атрибута, которые могут быть получены функцией mq_getattr и установлены (по отдельности) функцией mq_setattr:

Структура mq_attr хранит в себе эти четыре атрибута:

Указатель на такую структуру может быть передан в качестве четвертого аргумента mq_open, что дает возможность установить параметры mq_maxmsg и mq_msgsize в момент создания очереди. Другие два поля структуры функцией mq_open игнорируются.

Функция mq_getattr присваивает полям структуры, на которую указывает *attr*, текущие значения атрибутов очереди.

Функция mq_setattr устанавливает атрибуты очереди, но фактически используется только поле mqflags той структуры, на которую указывает *attr*, что дает возможность сбрасывать или устанавливать флаг запрета блокировки. Другие три поля структуры игнорируются: максимальное количество сообщений в очереди и максимальный размер сообщения могут быть установлены только в момент создания очереди, а количество сообщений в очереди можно только считать, но не изменить.

Кроме того, если указатель *oattr* ненулевой, возвращаются предыдущие значения атрибутов очереди (mq_flags, mq_maxmsg, mq_msgsize) и текущий статус очереди (mq_curmsgs).

Пример: программа mqgetattr

Программа из листинга 5.3 открывает указанную очередь сообщений и выводит значения ее атрибутов.

Мы можем создать очередь сообщений и вывести значения ее атрибутов, устанавливаемые по умолчанию:

Вспомним размер одного из файлов очереди, созданной с использованием устанавливаемых по умолчанию значений атрибутов. Он был выведен командой ls в примере после листинга 5.1. Это значение можно получить как $128 \times 1024 + 1560 = 132632$.

Добавочные 1560 байт представляют собой, скорее всего, дополнительную информацию: 8 байт на сообщение плюс добавочные 536 байт.

Пример: программа mqcreate

Мы можем изменить программу из листинга 5.1 таким образом, чтобы при создании очереди иметь возможность указывать максимальное количество сообщений и максимальный размер сообщения. Мы не можем указать только один из этих параметров; нужно обязательно задать оба (см., впрочем, упражнение 5.1). В листинге 5.4 приведен текст новой программы.

Параметр командной строки, требующий аргумента, указывается с помощью двоеточия (после параметров *m* и *z* в вызове getopt). В момент обработки символа параметр optarg указывает на аргумент.

ПРИМЕЧАНИЕ

Наша обертка Getopt вызывает стандартную библиотечную функцию getopt и завершает выполнение процесса в случае возникновения ошибок в ее работе: при появлении параметра, не указанного в третьем аргументе при вызове функции, или при наличии параметра без необходимого числового аргумента (потребность в нем указывается с помощью двоеточия после буквы параметра в третьем аргументе функции getopt). В любом случае, getopt помещает сообщение об ошибке в стандартный поток сообщений об ошибках и возвращает ошибку, что приводит к завершению работы оберткой Getopt. В двух приведенных ниже примерах ошибка обнаруживается функцией getopt:

В следующем примере ошибка (не указан необходимый аргумент — имя очереди) обнаруживается самой программой:

Если не указан ни один из двух новых параметров, мы должны передать функции mq_open пустой указатель в качестве последнего аргумента. В противном случае мы передаем указатель на нашу структуру attr.

Запустим теперь новую версию нашей программы в системе Solaris 2.6, указав максимальное количество сообщений 1024 и максимальный размер сообщения 8192 байт:

Размер файла, содержащего данные этой очереди, соответствует максимальному количеству сообщений в очереди и максимальному размеру сообщения ($1024 \times 8192 = 8388608$), а оставшиеся 8728 байт предусматривают 8 байт информации на каждое сообщение (8×1024) плюс дополнительные 536 байт.

При выполнении той же программы в Digital Unix 4.0B получим:

В этой реализации размер очереди соответствует максимальному количеству сообщений и максимальному размеру сообщения ($256 \times 2048 = 524288$), а оставшиеся 13000 байт дают возможность хранить 48 байт добавочной информации для каждого сообщения (48×256) и еще 712 байт.

Эти две функции предназначены для помещения сообщений в очередь и получения их оттуда. Каждое сообщение имеет свой приоритет, который представляет собой беззнаковое целое, не превышающее MQ_PRIO_MAX. Стандарт Posix требует, чтобы эта величина была не меньше 32.

ПРИМЕЧАНИЕ

В Solaris 2.6 значение MQ_PRIO_MAX равняется именно 32, но в Digital Unix 4.0B этот предел равен уже 256. В листинге 5.7 мы покажем, как получить эти значения.

Функция mq_receive всегда возвращает старейшее в указанной очереди сообщение с максимальным приоритетом, и приоритет может быть получен вместе с содержимым сообщения и его длиной.

ПРИМЕЧАНИЕ

Действие mq_receive отличается от действия msgrcv в System V (раздел 6.4). Сообщения System V имеют поле type, аналогичное по смыслу приоритету, но для функции msgrcv можно указать три различных алгоритма возвращения сообщений: старейшее сообщение в очереди, старейшее сообщение с указанным типом или старейшее сообщение с типом, не превышающим указанного значения.

Первые три аргумента обеих функций аналогичны первым трем аргументам функций write и read соответственно.

ПРИМЕЧАНИЕ

Объявление указателя на буфер как char* кажется ошибкой — тип void* больше соответствовал бы по духу прочим функциям Posix.1.

Значение аргумента len функции mq_receive должно быть по крайней мере не меньше максимального размера сообщения, которое может быть помещено в очередь, то есть значения поля mq_msgsize структуры mq_attr для этой очереди. Если len оказывается меньше этой величины, немедленно возвращается ошибка EMSGSIZE.

ПРИМЕЧАНИЕ

Это означает, что большинству приложений, использующих очереди сообщений Posix, придется вызывать mq_getattr после открытия очереди для определения максимального размера сообщения, а затем выделять память под один или несколько буферов чтения этого размера. Требование, чтобы буфер был больше по размеру, чем максимально возможное сообщение, позволяет функции mq_receive не возвращать уведомление о том, что размер письма превышает объем буфера. Сравните это, например, с флагом MSG_NOERROR и ошибкой E2BIG для очередей сообщений System V (раздел 6.4) и флагом MSG_TRUNC для функции recvmsg, используемой с дейтаграммами UDP (раздел 13.5 [24]).

Аргумент prio устанавливает приоритет сообщения для mq_send, его значение должно быть меньше MQ_PRIO_MAX. Если при вызове mq_receive prior является ненулевым указателем, в нем сохраняется приоритет возвращаемого сообщения. Если приложению не требуется использование различных приоритетов сообщений, можно указывать его равным нулю для mq_send и передавать mq_receive нулевой указатель в качестве последнего аргумента.

ПРИМЕЧАНИЕ

Разрешена передача сообщений нулевой длины. Это тот случай, когда важно не то, о чём говорится в стандарте (Posix.1), а то, о чём в нем не говорится: нигде не запрещена передача сообщений нулевой длины. Функция mq_receive возвращает количество байтов в сообщении (в случае успешного завершения работы) или -1 в случае возникновения ошибок, так что 0 обозначает сообщение нулевой длины.

Очередям сообщений Posix и System V не хватает полезной функции: получатель не может определить отправителя сообщения. Эта информация могла бы пригодиться многим приложениям. К сожалению, большинство механизмов передачи сообщений IPC не позволяют определить отправителя сообщений. В разделе 15.5 мы расскажем, как эта возможность обеспечивается для дверей. В разделе 14.8 [24] описано, как эта возможность обеспечивается в BSD/OS для доменных сокетов Unix. В разделе 15.3.1 [21] описано, как SVR4 передает информацию об отправителе по каналу при передаче по нему дескриптора. В настоящее время методы BSD/OS широко используются, и хотя реализация SVR4 является частью стандарта Unix 98, она требует передачи дескриптора по каналу, что обычно является более дорогостоящей операцией, чем просто передача данных. Мы не можем предоставить отправителю возможность передать информацию о себе (например, эффективный идентификатор пользователя) в самом сообщении, поскольку мы не можем быть уверены, что эта информация окажется истинной. Хотя разрешения доступа к очереди

сообщений определяют, имеет ли право отправитель помещать в нее сообщения, это все равно не дает однозначности. Существует возможность создавать одну очередь для каждого отправителя (о которой рассказывается в связи с очередями System V в разделе 6.8), но это плохо подходит для больших приложений. Наконец, если функции для работы с очередями сообщений реализованы как пользовательские функции (как мы показываем в разделе 5.8), а не как часть ядра, мы не можем доверять никакой информации об отправителе, передаваемой с сообщением, так как ее легко подделать.

Пример: программа mqsend

В листинге 5.5 приведен текст программы, помещающей сообщение в очередь.

И размер сообщения, и его приоритет являются обязательными аргументами командной строки. Буфер под сообщение выделяется функцией calloc, которая инициализирует его нулем.

Пример: программа mqreceive

Программа в листинге 5.6 считывает сообщение из очереди.

14-17 Параметр командной строки -n отключает блокировку. При этом программа возвращает сообщение об ошибке, если в очереди нет сообщений.

21-25 Мы открываем очередь и получаем ее атрибуты, вызвав mq_getattr. Нам обязательно нужно определить максимальный размер сообщения, потому что мы должны выделить буфер подходящего размера, чтобы вызвать mq_receive. Программа выводит размер считываемого сообщения и его приоритет.

ПРИМЕЧАНИЕ

Поскольку n имеет тип size_t и мы не знаем, int это или long, мы преобразуем эту величину к типу long и используем строку формата %ld. В 64-разрядной реализации int будет 32-разрядным целым, а long и size_t будут 64-разрядными целыми.

Воспользуемся обеими программами, чтобы проиллюстрировать использование поля приоритета.

Мы видим, что mq_receive действительно возвращает старейшее сообщение с наивысшим приоритетом.

Мы уже сталкивались с двумя ограничениями, устанавливаемыми для любой очереди в момент ее создания:

- mq_maxmsg — максимальное количество сообщений в очереди;
- mq_msgsize — максимальный размер сообщения.

Не существует каких-либо ограничений на эти значения, хотя в рассматриваемых реализациях необходимо наличие в файловой системе места для файла требуемого размера. Кроме того, ограничения на эти величины могут накладываться реализацией виртуальной памяти (см. упражнение 5.5).

Другие два ограничения определяются реализацией:

- MQ_OPEN_MAX — максимальное количество очередей сообщений, которые могут быть одновременно открыты каким-либо процессом (Posix требует, чтобы эта величина была не меньше 8);
- MQ_PRIO_MAX — максимальное значение приоритета плюс один (Posix требует, чтобы эта величина была не меньше 32).

Эти две константы часто определяются в заголовочном файле `<unistd.h>` и могут быть получены во время выполнения программы вызовом функции `sysconf`, как мы покажем далее.

Пример: программа mqsysconf

Программа в листинге 5.7 вызывает функцию sysconf и выводит два ограничения на очереди сообщений, определяемые реализацией.

Запустив эту программу в наших двух операционных системах, получим:

Один из недостатков очередей сообщений System V, как мы увидим в главе 6, заключается в невозможности уведомить процесс о том, что в очередь было помещено сообщение. Мы можем заблокировать процесс при вызове `msgrecv`, но тогда мы не сможем выполнять другие действия во время ожидания сообщения. Если мы укажем флаг отключения блокировки при вызове `msgrecv` (`IPC_NOWAIT`), процесс не будет заблокирован, но нам придется регулярно вызывать эту функцию, чтобы получить сообщение, когда оно будет отправлено. Мы уже говорили, что такая процедура называется *опросом* и на нее тратится лишнее время. Нужно, чтобы система сама уведомляла процесс о том, что в пустую очередь было помещено новое сообщение.

ПРИМЕЧАНИЕ

В этом и всех последующих разделах данной главы обсуждаются более сложные вопросы, которые могут быть пропущены при первом чтении.

Очереди сообщений Posix допускают асинхронное уведомление о событии, когда сообщение помещается в очередь. Это уведомление может быть реализовано либо отправкой сигнала, либо созданием программного потока для выполнения указанной функции.

Мы включаем режим уведомления с помощью функции `mq_notify`:

Эта функция включает и выключает асинхронное уведомление о событии для указанной очереди. Структура `sigevent` впервые появилась в стандарте Posix.1 для сигналов реального времени, о которых более подробно рассказано в следующем разделе. Эта структура и все новые константы, относящиеся к сигналам, определены в заголовочном файле `<signal.h>`:

Мы вскоре приведем несколько примеров различных вариантов использования уведомления, но о правилах, действующих для этой функции всегда, можно упомянуть уже сейчас.

1. Если аргумент *notification* ненулевой, процесс ожидает уведомления при поступлении нового сообщения в указанную очередь, пустую на момент его поступления. Мы говорим, что процесс *регистрируется на уведомление для данной очереди*.
2. Если аргумент *notification* представляет собой нулевой указатель и процесс уже зарегистрирован на уведомление для данной очереди, то уведомление для него отключается.
3. Только один процесс может быть зарегистрирован на уведомление для любой данной очереди в любой момент.
4. При помещении сообщения в пустую очередь, для которой имеется зарегистрированный на уведомление процесс, оно будет отправлено только в том случае, если нет заблокированных в вызове `mq_receive` для этой очереди процессов. Таким образом, блокировка в вызове `mq_receive` имеет приоритет перед любой регистрацией на уведомление.
5. При отправке уведомления зарегистрированному процессу регистрация снимается. Процесс должен зарегистрироваться снова (если в этом есть необходимость), вызвав `mq_notify` еще раз.

ПРИМЕЧАНИЕ

С сигналами в Unix всегда была связана одна проблема: действие сигнала сбрасывалось на установленное по умолчанию каждый раз при отправке сигнала (раздел 10.4 [21]). Обычно первой функцией, вызываемой обработчиком сигнала, была `signal`, переопределявшая обработчик. Это создавало небольшой временной промежуток между отправкой сигнала и переопределением обработчика, в который процесс мог быть завершен при повторном появлении того же сигнала. На первый взгляд может показаться, что та же проблема должна возникать и при использовании `mq_notify`, поскольку процесс должен перерегистрироваться каждый раз после появления уведомления. Однако очереди сообщений отличаются по своим свойствам от сигналов, поскольку необходимость отправки уведомления не может возникнуть, пока очередь не будет пуста. Следовательно, необходимо аккуратно перерегистрироваться на получение уведомления до считывания пришедшего сообщения из очереди.

Пример: простая программа с уведомлением

Прежде чем углубляться в тонкости сигналов реального времени и потоков Posix, мы напишем простейшую программу, включающую отправку сигнала SIGUSR1 при помещении сообщения в пустую очередь. Эта программа приведена в листинге 5.8, и мы отметим, что она содержит ошибку, о которой мы вскоре поговорим подробно.

2-6 Мы объявляем несколько глобальных переменных, используемых совместно функцией main и нашим обработчиком сигнала (sig_usr1).

12-15 Мы открываем очередь сообщений, получаем ее атрибуты и выделяем буфер считывания соответствующего размера.

16-20 Сначала мы устанавливаем свой обработчик для сигнала SIGUSR1. Мы присваиваем полю sigev_notify структуры sigevent значение SIGEV_SIGNAL, что говорит системе о необходимости отправки сигнала, когда очередь из пустой становится непустой. Полю sigev_signo присваивается значение, соответствующее тому сигналу, который мы хотим получить. Затем вызывается функция mq_notify.

Функция main после этого зацикливается, и процесс приостанавливается при вызове pause, возвращающей -1 при получении сигнала.

Обработчик сигнала вызывает mq_notify для перерегистрации, считывает сообщение и выводит его длину. В этой программе мы игнорируем приоритет полученного сообщения.

ПРИМЕЧАНИЕ

Оператор return в конце sig_usr1 не требуется, поскольку возвращаемое значение отсутствует, а конец текста функции неявно предусматривает возвращение в вызвавшую программу. Тем не менее автор всегда записывает return явно, чтобы указать, что возвращение из этой функции может происходить с особенностями. Например, может произойти преждевременный возврат (с ошибкой EINTR) в потоке, обрабатывающем сигнал.

Запустим теперь эту программу в одном из окон

и затем выполним следующую команду в другом окне

Как и ожидалось, программа mqnotifysig1 выведет сообщение: SIGUSR1 received, read 50 bytes.

Мы можем проверить, что только один процесс может быть зарегистрирован на получение уведомления в любой момент, запустив копию программы в другом окне:

Это сообщение соответствует коду ошибки EBUSY.

Сигналы Posix: функции типа Async-Signal-Safe

Недостаток программы из листинга 5.8 в том, что она вызывает `mq_notify`, `mq_receive` и `printf` из обработчика сигнала. Ни одну из этих функций вызывать оттуда не следует.

Функции, которые могут быть вызваны из обработчика сигнала, относятся к группе, называемой, согласно Posix, *async-signal-safe functions* (функции, обеспечивающие безопасную обработку асинхронных сигналов). В табл. 5.1 приведены эти функции по стандарту Posix вместе с некоторыми дополнительными, появившимися только в Unix 98.

Функции, которых нет в этом списке, не должны вызываться из обработчика сигнала. Обратите внимание, что в списке отсутствуют стандартные функции библиотеки ввода-вывода и функции `pthread_XXX` для работы с потоками. Из всех функций IPC, рассматриваемых в этой книге, в список попали только `sem_post`, `read` и `write` (подразумевается, что последние две используются с программными каналами и FIFO).

ПРИМЕЧАНИЕ

Стандарт ANSI C указывает четыре функции, которые могут быть вызваны из обработчика сигналов: `abort`, `exit`, `longjmp`, `signal`. Первые три отсутствуют в списке функций *async-signal-safe* стандарта Unix 98.

Таблица 5.1. Функции, относящиеся к группе *async-signal-safe*

Пример: уведомление сигналом

Одним из способов исключения вызова каких-либо функций из обработчика сигнала является установка этим обработчиком глобального флага, который проверяется программным потоком для получения информации о приходе сообщения. В листинге 5.9 иллюстрируется этот метод, хотя новая программа также содержит ошибку, но уже другую, о которой мы вскоре поговорим подробнее.

2 Поскольку единственное действие, выполняемое обработчиком сигнала, заключается в присваивании ненулевого значения флагу mqflag, глобальным переменным из листинга 5.8 уже не нужно являться таковыми. Уменьшение количества глобальных переменных — это всегда благо, особенно при использовании программных потоков.

15-18 Мы открываем очередь сообщений, получаем ее атрибуты и выделяем буфер считывания.

19-22 Мы инициализируем три набора сигналов и устанавливаем бит для сигнала SIGUSR1 в наборе newmask.

23-27 Мы устанавливаем обработчик сигнала для SIGUSR1, присваиваем значения полям структуры sigevent и вызываем mq_notify.

28-32 Мы вызываем sigprocmask, чтобы заблокировать SIGUSR1, сохраняя текущую маску сигналов в oldmask. Затем мы в цикле проверяем значение глобального флага mqflag, ожидая, когда обработчик сигнала установит его в ненулевое значение. Пока значение этого флага равно нулю, мы вызываем sigsuspend, что автоматически приостанавливает вызывающий поток и устанавливает его маску в zeromask (сигналы не блокируются). Раздел 10.16 [21] рассказывает о функции sigsuspend более подробно. Также там объясняются причины, по которым мы должны проверять значение переменной mqflag только при заблокированном сигнале SIGUSR1. Каждый раз при выходе из sigsuspend сигнал SIGUSR1 блокируется.

33-36 Когда флаг mqflag принимает ненулевое значение, мы регистрируемся на получение уведомления заново и считываем сообщение из очереди. Затем мы разблокируем сигнал SIGUSR1 и возвращаемся к началу цикла.

Мы уже говорили, что в этой версии программы также присутствует ошибка. Посмотрим, что произойдет, если в очередь попадут два сообщения, прежде чем будет считано первое из них. Мы можем имитировать это, добавив sleep перед вызовом mq_notify. Проблема тут в том, что уведомление отсылается только в том случае, когда сообщение помещается в пустую очередь. Если в очередь поступают два сообщения, прежде чем первое будет считано, то отсылается только одно уведомление. Тогда мы считываем первое сообщение и вызываем sigsuspend, ожидая поступления еще одного. А в это время в очереди уже имеется сообщение, которое мы должны прочитать, но которое мы никогда не прочтем.

Пример: уведомление сигналом с отключением блокировки

Исправить описанную выше ошибку можно, отключив блокировку операции считывания сообщений. Листинг 5.10 содержит измененную версию программы из листинга 5.9. Новая программа считывает сообщения в неблокируемом режиме.

15-18 Первое изменение в программе: при открытии очереди сообщений указывается флаг O_NONBLOCK.

34-38 Другое изменение: mq_receive вызывается в цикле, считывая все сообщения в очереди, пока не будет возвращена ошибка с кодом EAGAIN, означающая отсутствие сообщений в очереди.

Пример: уведомление с использованием sigwait вместо обработчика

Хотя программа из предыдущего примера работает правильно, можно повысить ее эффективность. Программа использует sigsuspend для блокировки в ожидании прихода сообщения. При помещении сообщения в пустую очередь вызывается сигнал, основной поток останавливается, запускается обработчик, который устанавливает флаг mqflag, затем снова запускается главный поток, он обнаруживает, что значение mqflag отлично от нуля, и считывает сообщение. Более простой и эффективный подход заключается в блокировании в функции, ожидающей получения сигнала, что не требует вызова обработчика только для установки флага. Эта возможность предоставляется функцией sigwait:

Перед вызовом sigwait мы блокируем некоторые сигналы. Набор блокируемых сигналов указывается в качестве аргумента set. Функция sigwait блокируется, пока не придет по крайней мере один из этих сигналов. Когда он будет получен, функция возвратит его. Значение этого сигнала сохраняется в указателе *sig*, а функция возвращает значение 0. Это называется *синхронным ожиданием асинхронного события*: мы используем сигнал, но не пользуемся асинхронным обработчиком сигнала.

В листинге 5.11 приведен текст программы, использующей mq_notify и sigwait.

18-20 Инициализируется один набор сигналов, содержащий только SIGUSR1, а затем этот сигнал блокируется sigprocmask.

26-34 Мы блокируем выполнение программы и ждем прихода сигнала, вызвав sigwait. При получении сигнала SIGUSR1 мы перерегистрируемся на уведомление и считываем все доступные сообщения.

ПРИМЕЧАНИЕ

Функция sigwait часто используется в многопоточных процессах. Действительно, глядя на прототип функции, мы можем заметить, что возвращаемое значение будет 0 или одной из ошибок Exxx, что весьма похоже на функции Pthread. Однако в многопоточном процессе нельзя пользоваться sigprocmask — вместо нее следует вызывать pthread_sigmask, которая изменяет маску сигналов только для вызвавшего ее потока. Аргументы pthread_sigmask совпадают с аргументами sigprocmask.

Существуют два варианта функции sigwait: sigwaitinfo возвращает структуру siginfo_t (которая будет определена в следующем разделе) и предназначена для использования с надежными сигналами; функция sigtimedwait также возвращает структуру siginfo_t и позволяет вызывающему процессу установить ограничение по времени на ожидание.

Большая часть книг о многопоточном программировании, таких как [3], рекомендуют пользоваться sigwait для обработки всех сигналов в многопоточном процессе и не использовать асинхронные обработчики.

Пример: очереди сообщений Posix и функция select

Дескриптор очереди сообщений (переменная типа mqd_t) не является «обычным» дескриптором и не может использоваться с функциями select и poll (глава 6 [24]). Тем не менее их можно использовать вместе с каналом и функцией mq_notify. (Аналогичный метод применен в разделе 6.9 для очередей System V, где создается дочерний процесс и канал связи.) Прежде всего обратите внимание, что, согласно табл. 5.1, функция write принадлежит к группе async-signal-safe, поэтому она может вызываться из обработчика сигналов. Программа приведена в листинге 5.12.

21 Мы создаем канал, в который обработчик сигнала произведет запись, когда будет получено уведомление о поступлении сообщения в очередь. Это пример использования канала внутри одного процесса.

27-40 Мы инициализируем набор дескрипторов gset и при каждом проходе цикла включаем бит, соответствующий дескриптору pipefd[0] (открытый на считывание конец канала). Затем мы вызываем функцию select, ожидая получения единственного дескриптора, хотя в типичном приложении именно здесь осуществлялось бы размножение дескрипторов одного из концов канала. Когда появляется возможность читать из канала, мы перерегистрируемся на уведомление и считываем все доступные сообщения.

43-48 Единственное, что делает обработчик сигнала, — записывает в канал 1 байт. Как мы уже отмечали, эта операция относится к разрешенным для асинхронных обработчиков.

Пример: запуск нового потока

Альтернативой снятию блокировки сигналом является присваивание `sigev_notify` значения `SIGEV_THREAD`, что приводит к созданию нового потока. Функция, указанная в `sigev_notify_function`, вызывается с параметром `sigev_value`. Атрибуты нового канала указываются переменной `sigev_notify_attributes`, которая может быть и нулевым указателем, если нас устраивают устанавливаемые по умолчанию атрибуты. Текст программы приведен в листинге 5.13.

Мы задаем нулевой указатель в качестве аргумента нового потока (`sigev_value`), поэтому функции `start` нового потока ничего не передается. Мы могли бы передать указатель на дескриптор, вместо того чтобы декларировать его как глобальный, но новому потоку все равно нужно получать атрибуты очереди сообщений и структуру `sigev` (для перерегистрации). Мы также указываем нулевой указатель в качестве атрибутов нового потока, поэтому используются установки по умолчанию. Новые потоки создаются как неприсоединенные (detached threads).

ПРИМЕЧАНИЕ

К сожалению, ни одна из использовавшихся для проверки примеров систем (Solaris 2.6 и Digital Unix 4.0B) не поддерживает `SIGEV_THREAD`. Обе они допускают только два значения `sigev_notify`: `SIGEV_NONE` и `SIGEV_SIGNAL`.

За прошедшие годы сигналы в Unix много раз претерпевали революционные изменения.

1. Модель сигналов, предлагавшаяся в Unix Version 7 (1978), была ненадежной. Сигналы могли быть потеряны, и процессу было трудно отключить отдельные сигналы при выполнении отдельных участков кода.
2. В версии 4.3BSD (1986) надежные сигналы были добавлены.
3. Версия System V Release 3.0 (1986) также добавила надежные сигналы, хотя и иначе, чем BSD.
4. Стандарт Posix.1 (1990) увековечил модель надежных сигналов BSD, и эта модель подробно описана в главе 10 [21].
5. Posix.1 (1996) добавил к модели Posix сигналы реального времени. Это произросло из расширений реального времени Posix.1b (которые были названы Posix.4).

Почти все системы Unix в настоящее время поддерживают надежные сигналы, а новейшие системы предоставляют также и сигналы реального времени стандарта Posix. (Следует различать надежные сигналы и сигналы реального времени.) О сигналах реального времени следует поговорить подробнее, поскольку мы уже столкнулись с некоторыми структурами, определяемыми этим расширением стандарта, в предыдущем разделе (структуры sigval и sigevent).

Сигналы могут быть отнесены к двум группам:

1. Сигналы реального времени, которые могут принимать значения между SIGRTMIN и SIGRTMAX включительно. Posix требует, чтобы предоставлялось по крайней мере RTSIG_MAX сигналов, и минимальное значение этой константы равно 8.
2. Все прочие сигналы: SIGALRM, SIGINT, SIGKILL и пр.

ПРИМЕЧАНИЕ

В Solaris 2.6 обычные сигналы Unix нумеруются с 1 по 37, а 8 сигналов реального времени имеют номера с 38 по 45. В Digital Unix 4.0B обычные сигналы нумеруются с 1 по 32, а 16 сигналов реального времени имеют номера с 33 по 48. Обе реализации определяют SIGRTMIN и SIGRTMAX как макросы, вызывающие sysconf, что позволяет изменять их значения.

Далее все зависит от того, установлен ли процессом, получившим сигнал, флаг SA_SIGINFO при вызове sigaction. В итоге получаются четыре возможных сценария, приведенных в табл. 5.2.

Таблица 5.2. Поведение сигналов Posix в реальном времени в зависимости от SA_SIGINFO

Смысл фразы «характеристики реального времени не обязательны» следующий: некоторые реализации могут обрабатывать эти сигналы как сигналы реального времени, но это не обязательно. Если мы хотим, чтобы сигналы обрабатывались как сигналы реального времени, мы должны использовать сигналы с номерами от SIGRTMIN до SIGRTMAX и должны указать флаг SA_SIGINFO при вызове sigaction при установке обработчика сигнала.

Термин «характеристики реального времени» подразумевает следующее:

- Сигналы помещаются в очередь. Если сигнал будет порожден трижды, он будет трижды получен адресатом. Более того, повторения одного и того же сигнала доставляются в порядке очереди (FIFO). Мы вскоре покажем пример очереди сигналов. Если же сигналы в очередь не помещаются, трижды порожденный сигнал будет получен лишь один раз.
- Когда в очередь помещается множество неблокируемых сигналов в диапазоне SIGRTMIN—SIGRTMAX, сигналы с меньшими номерами доставляются раньше сигналов с большими номерами. То есть сигнал с номером SIGRTMIN имеет «больший приоритет», чем сигнал с номером SIGRTMIN+1, и т.д.
- При отправке сигнала, не обрабатываемого как сигнал реального времени, единственным аргументом обработчика является номер сигнала. Сигналы реального времени несут больше информации, чем прочие сигналы. Обработчик для сигнала реального времени, устанавливаемый с флагом SA_SIGINFO, объявляется как

где *signo* — номер сигнала, а *siginfo_t* — структура, определяемая как

На что указывает *context* — зависит от реализации.

ПРИМЕЧАНИЕ

Обработчик сигналов, не являющихся сигналами реального времени, вызывается с единственным аргументом. Во многих системах существует старое соглашение о вызове обработчиков сигналов с тремя аргументами, которое предшествовало стандарту реального времени Posix.

Тип `siginfo_t` является единственной структурой Posix, определяемой оператором `typedef` с именем, оканчивающимся на `_t`. В листинге 5.14 мы объявляем указатели на эти структуры как `siginfo_t *` без слова `struct`.

■ Для работы с сигналами реального времени добавлено несколько новых функций. Например, для отправки сигнала какому-либо процессу используется функция `sigqueue` вместо `kill`. Новая функция позволяет отправить вместе с сигналом структуру `sigval`.

Сигналы реального времени порождаются нижеследующими функциями Posix.1, определяемыми значением `si_code`, которое хранится в структуре `siginfo_t`, передаваемой обработчику сигнала.

- `SI_ASYNCIO` — сигнал был порожден по завершении асинхронного запроса на ввод или вывод одной из функций Posix `aio_XXX`, которые мы не рассматриваем;
- `SI_MESGQ` — сигнал был порожден при помещении сообщения в пустую очередь сообщений (как в разделе 5.6);
- `SI_QUEUE` — сигнал был отправлен функцией `sigqueue`. Пример будет вскоре приведен;
- `SI_TIMER` — сигнал был порожден по истечении установленного функцией `timer_settime` времени. Этую функцию мы не описываем;
- `SI_USER` — сигнал был отправлен функцией `kill`.

Если сигнал был порожден каким-либо другим событием, `si_code` будет иметь значение, отличающееся от приведенных выше. Значение поля `si_value` структуры `siginfo_t` актуально только в том случае, если `si_code` имеет одно из следующих значений: `SI_ASYNCIO`, `SI_MESGQ`, `SI_QUEUE` и `SI_TIMER`.

Пример

В листинге 5.14 приведен пример программы, демонстрирующей использование сигналов реального времени. Программа вызывает fork, дочерний процесс блокирует три сигнала реального времени, родительский процесс отправляет девять сигналов (три раза отсылается каждый из заблокированных сигналов), затем дочерний процесс разблокирует сигналы и мы смотрим, сколько раз будет получен каждый из них и в каком порядке они придут.

10 Мы печатаем наибольший и наименьший номера сигналов реального времени, чтобы узнать, сколько их предоставляетяется в данной реализации. Мы преобразуем обе константы к типу integer, поскольку в некоторых реализациях они определяются как макросы, требующие вызова sysconf, например:

и функция sysconf возвращает целое типа long (см. упражнение 5.4).

11-17 Запускается дочерний процесс, который вызывает sigprocmask для блокировки трех используемых сигналов реального времени: SIGRTMAX, SIGRTMAX-1 и SIGRTMAX-2.

18-21 Мы вызываем функцию signal_rt (приведенную в листинге 5.15) для установки функции sig_rt в качестве обработчика трех указанных выше сигналов реального времени. Функция устанавливает флаг SA_SIGINFO, и поскольку эти три сигнала являются сигналами реального времени, мы можем ожидать, что они будут обрабатываться соответствующим образом. Эта функция также устанавливает маску сигналов, блокируемых на время выполнения обработчика.

22-25 Дочерний процесс ждет 6 секунд, пока родительский породит девять сигналов. Затем вызывается sigprocmask для разблокирования трех сигналов реального времени. Это позволяет всем помещенным в очередь сигналам достичь адресата. Затем делается пауза еще на три секунды, чтобы обработчик успел вызвать printf девять раз, после чего дочерний процесс завершает свою работу.

27-36 Родительский процесс ждет три секунды, пока дочерний не заблокирует все требуемые сигналы. Затем родительский процесс порождает три экземпляра каждого из трех сигналов реального времени: i принимает 3 значения, а j принимает значения 0, 1 и 2 для каждого из значений i. Мы преднамеренно порождаем сигналы начиная с наибольшего номера, поскольку ожидаем, что они будут получены начиная с наименьшего. Мы также отсылаем с каждым из сигналов новое значение sigval_int, чтобы проверить, что копии одного и того же сигнала доставляются в том же порядке, в каком они были отправлены, то есть очередь действительно является очередью.

38-43 Обработчик сигнала просто выводит информацию о полученном сигнале.

ПРИМЕЧАНИЕ

Из табл. 5.1 следует, что функция printf не относится к функциям типа async-signal-safe и не должна вызываться из обработчика сигналов. Здесь мы используем ее исключительно в качестве проверочного средства в маленькой тестовой программе.

Запустим эту программу в Solaris 2.6. Результат будет не тем, которого мы ожидали:

В очередь помещаются девять сигналов, но первыми принимаются сигналы с большими номерами (а мы ожидали получить сигналы с меньшими номерами).

Кроме того, сигналы с одинаковым номером приходят в порядке LIFO, а не FIFO. Код si_code = -2 соответствует SI_QUEUE.

Запустив программу в Digital Unix 4.0B, мы получим именно тот результат, которого ожидали:

Девять сигналов помещаются в очередь и получаются адресатом в ожидаемом порядке: первым приходит сигнал с меньшим номером, а копии сигнала приходят в порядке FIFO.

ПРИМЕЧАНИЕ

Похоже, что в реализации Solaris 2.6 есть ошибка.

Функция signal_rt

В книге [24, с. 120] мы привели пример собственной функции signal, вызывавшей функцию sigaction стандарта Posix для установки обработчика сигнала, обеспечивающего надежную семантику Posix. Изменим эту функцию, чтобы обеспечить поддержку реального времени. Новую функцию мы назовем signal_rt; ее текст приведен в листинге 5.15.

1-3 В нашем заголовочном файле unprirc.h (листинг B.1) мы определяем Sigfunc_rt как

Ранее в этом разделе мы говорили о том, что это прототип функции для обработчика сигнала, устанавливаемого с флагом SA_SIGINFO.

Структура sigaction претерпела изменения с добавлением поддержки сигна-5-7 лов реального времени: к ней было добавлено новое поле sa_sigaction:

Правила действуют следующие:

- Если в поле sa_flags установлен флаг SA_SIGINFO, поле sa_sigaction указывает адрес функции-обработчика сигнала.
- Если флаг SA_SIGINFO не установлен, поле sa_handler указывает адрес функции-обработчика сигнала.
- Чтобы сопоставить сигналу действие по умолчанию или игнорировать его, следует установить sa_handler равным либо SIG_DFL, либо SIG_IGN и не устанавливать флаг SA_SIGINFO.

8-17 Мы всегда устанавливаем флаг SA_SIGINFO и указываем флаг SA_RESTART, если перехвачен какой-либо другой сигнал, кроме SIGALRM.

Теперь рассмотрим реализацию очередей сообщений Posix с использованием отображения в память, взаимных исключений и условных переменных Posix.

ПРИМЕЧАНИЕ

Взаимные исключения и условные переменные описаны в главе 7, а ввод-вывод с отображением в память — в главах 12 и 13. Вы можете отложить данный раздел до ознакомления с этими главами.

На рис. 5.2 приведена схема структур данных, которыми мы пользуемся для реализации очереди сообщений Posix. Изображенная очередь может содержать до четырех сообщений по 7 байт каждое.

В листинге 5.16 приведен заголовочный файл mqqueue.h, определяющий основные структуры, используемые в этой реализации.

Дескриптор нашей очереди сообщений является просто указателем на структуру mq_info. Каждый вызов mq_open выделяет память под одну такую структуру, а вызвавшему возвращается указатель на нее. Повторим, что дескриптор очереди сообщений не обязательно является небольшим целым числом, как дескриптор файла — единственное ограничение, накладываемое Posix, заключается в том, что этот тип данных не может быть массивом.



Рис. 5.2. Схема структур данных, используемых при реализации очередей сообщений posix через отображаемый в память файл

8-18 Эта структура хранится в самом начале отображаемого файла и содержит всю информацию об очереди. Поле mq_flags структуры mqh_attr не используется, поскольку флаги (единственный определенный флаг используется для отключения блокировки) должны обрабатываться для каждого открывающего очередь процесса в отдельности, а не для очереди в целом. Поэтому флаги хранятся в структуре mq_info. О прочих полях этой структуры мы будем говорить в связи с их использованием различными функциями.

Обратите внимание, что все переменные, называемые нами индексными (поля этой структуры mqh_head и mqh_free, а также поле msg_next следующей структуры), содержат индексы байтов относительно начала отображаемого в память файла. Например, размер структуры mq_hdr в системе Solaris 2.6 — 96 байт, поэтому индекс первого сообщения, располагающегося сразу за заголовком, имеет значение 96. Каждое сообщение на рис. 5.2 занимает 20 байт (12 байт на структуру msg_hdr и 8 байт на данные), поэтому индексы следующих трех сообщений имеют значения 116, 136 и 156, а размер отображаемого в память файла — 176 байт. Индексы используются для обработки двух связных списков, хранящихся в этом файле: в одном из списков (mqh_head) хранятся все сообщения, имеющиеся в данный момент в очереди, а в другом (mqh_free) — все незаполненные сообщения. Мы не можем использовать настоящие указатели на области памяти (адреса) при работе со списком, поскольку отображаемый файл может находиться в произвольной области памяти для каждого из процессов, работающих с ним (как показано в листинге 13.5).

19-25 Эта структура располагается в начале каждого сообщения в отображаемом файле. Любое сообщение может принадлежать либо к списку заполненных, либо к списку свободных сообщений, и поле msg_next содержит индекс следующего сообщения в этом списке (или 0, если сообщение является в этом списке последним). Переменная msg_len хранит реальную длину данных в сообщении, которая в нашем примере с рис. 5.2 может иметь значение от 0 до 7 байт включительно. В переменную msg_prio отправителем помещается значение приоритета сообщения.

26-32 Экземпляр такой структуры динамически создается функцией mq_open при открытии очереди и удаляется mq_close. Поле mq_hdr указывает на отображаемый файл (адрес начала файла возвращается mmap). Указатель на эту структуру имеет основной в нашей реализации тип mqd_t, он принимает значение, возвращаемое mq_open.

Поле mq_magic принимает значение MQI_MAGIC в момент инициализации структуры. Это значение проверяется всеми функциями, которым передается указатель типа mqd_t, что дает им возможность удостовериться, что указатель действительно указывает на структуру типа mq_info.

`mqi_flags` содержит флаг отключения блокировки для открывшего очередь процесса.

33-34 В целях выравнивания содержимого файла (alignment) мы располагаем начало каждого сообщения так, чтобы его индекс был кратен размеру длинного целого. Следовательно, если максимальный размер сообщения не допускает такого выравнивания, мы добавляем к нему от 1 до 3 байт, как показано на рис. 5.2. При этом предполагается, что размер длинного целого — 4 байт (что верно для Solaris 2.6). Если размер длинного целого 8 байт (в Digital Unix 4.0B), нам придется добавлять к каждому сообщению от 1 до 7 байт.

Функция mq_open

В листинге 5.17 приведен текст первой части функции mq_open, создающей новую очередь сообщений или открываяющей существующую.

29-32 Функция может быть вызвана либо с двумя, либо с четырьмя аргументами в зависимости от того, указан ли флаг O_CREAT. Если флаг указан, третий аргумент имеет тип mode_t, а это простой системный тип, являющийся одним из целых типов. При этом мы столкнемся с проблемой в BSD/OS, где этот тип данных определен как unsigned short (16 бит). Поскольку целое в этой реализации занимает 32 бита, компилятор C увеличивает аргумент этого типа с 16 до 32 бит, потому что все короткие целые в списке аргументов увеличиваются до обычных целых. Но если мы укажем mode_t при вызове va_arg, он пропустит 16 бит аргумента в стеке, если этот аргумент был увеличен до 32 бит. Следовательно, мы должны определить свой собственный тип данных, va_mode_t, который будет целым в BSD/OS и типом mode_t в других системах. Эту проблему с переносимостью решают приведенные ниже строки нашего заголовка unprirc.h (листинг В.1):

30 Мы сбрасываем бит user-execute (S_IXUSR) в переменной mode по причинам, которые будут вскоре раскрыты.

33-34 Создается обычный файл с именем, указанным при вызове функции, и устанавливается бит user-execute.

35-40 Если бы при указании флага O_CREAT мы просто открыли файл, отобразили его содержимое в память и проинициализировали отображенный файл (как будет описано ниже), у нас возникла бы ситуация гонок. Очередь сообщений инициализируется mq_open только в том случае, если вызывающий процесс указывает флаг O_CREAT и очередь сообщений еще не существует. Это означает, что нам нужно каким-то образом определять, существует она или нет. Для этого при открытии файла для последующего отображения в память мы всегда указываем флаг O_EXCL. Возвращение ошибки EEXIST функцией open является ошибкой для mq_open только в том случае, если при вызове был указан флаг O_EXCL. В противном случае при возвращении функцией open ошибки EEXIST мы делаем вывод, что файл уже существует, и переходим к листингу 5.19, как если бы флаг O_CREAT вовсе не был указан.

Ситуация гонок может возникнуть потому, что использование отображаемого в память файла для реализации очереди сообщений требует двух шагов при инициализации очереди: сначала файл должен быть создан функцией open, а затем его содержимое должно быть проинициализировано. Проблема возникает, если два потока (одного или различных процессов) вызывают mq_open приблизительно одновременно. Один из потоков может создать файл, после чего управление будет передано системой второму потоку, прежде чем первый завершит инициализацию файла. Второй поток обнаружит, что файл уже существует (вызвав open с флагом O_EXCL), и приступит к использованию очереди сообщений.

Мы используем бит user-execute для указания того, был ли проинициализирован файл с очередью сообщений. Этот бит устанавливается только тем потоком, который создает этот файл (флаг O_EXCL позволяет определить этот поток); этот поток инициализирует файл с очередью сообщений, а затем сбрасывает бит user-execute.

Аналогичная ситуация может возникнуть в листингах 10.28 и 10.37.

42-50 Если при вызове в качестве последнего аргумента передан нулевой указатель, очередь сообщений инициализируется со значениями атрибутов по умолчанию: 128 сообщений в очереди и 1024 байта на сообщение. Если атрибуты указаны явно, мы проверяем, что mq_maxmsg и mq_msgsize имеют положительные значения.

Вторая часть функции mq_open приведена в листинге 5.18. Она завершает инициализацию новой очереди сообщений.

51-58 Вычисляется размер сообщения, который затем округляется до кратного размеру длинного целого. Также в файле отводится место для структуры mq_hdr в начале файла и msghdr в начале каждого сообщения (рис. 5.2). Размер вновь созданного файла устанавливается функцией lseek и записью одного байта со значением 0. Проще было бы вызвать ftruncate (раздел 13.3), но у нас нет гарантий, что это сработало бы для увеличения размера файла.

59-63 Файл отображается в память функцией mmap.

64-66 При каждом вызове mq_open создается отдельный экземпляр mq_info. Эта структура после создания инициализируется.

67-87 Инициализируется структура mq_hdr. Заголовок связного списка сообщений (mqh_head) инициализируется нулем, а все сообщения в очереди добавляются к списку свободных (mqh_free).

88-102 Поскольку очереди сообщений Posix могут использоваться совместно произвольным количеством процессов, которые знают имя очереди и имеют соответствующие разрешения, нам нужно инициализировать взаимное исключение и условную переменную с атрибутом PTHREAD_PROCESS_SHARED. Для этого мы сначала инициализируем атрибуты вызовом pthread_mutexattr_init, а затем устанавливаем значение атрибута совместного использования процессами, вызвав pthread_mutexattr_setpshared. После этого взаимное исключение инициализируется вызовом pthread_mutex_init. Аналогичные действия выполняются для условной переменной. Мы должны аккуратно удалить взаимное исключение и условную переменную даже при возникновении ошибки, поскольку вызовы pthread_mutexattr_init и pthread_condattr_init выделяют под них память (упражнение 7.3).

103-107 После инициализации очереди сообщений мы сбрасываем бит user-execute. Это говорит другим процессам о том, что очередь была проинициализирована. Мы также закрываем файл вызовом close, поскольку он был успешно отображен в память и держать его открытым больше нет необходимости.

В листинге 5.19 приведен конец функции mq_open, в котором осуществляется открытие существующей очереди сообщений.

109-115 Здесь мы завершаем работу, если флаг O_CREAT не был указан или если он был указан, но очередь уже существовала. В любом случае, мы открываем существующую очередь сообщений. Для этого мы открываем для чтения и записи файл, в котором она содержится, функцией open и отображаем его содержимое в адресное пространство процесса (mmap).

ПРИМЕЧАНИЕ

Наша реализация сильно упрощена в том, что касается режима открытия файла. Даже если вызвавший процесс указывает флаг O_RDONLY, мы должны дать возможность доступа для чтения и записи при открытии файла командой open и при отображении его в память командой mmap, поскольку невозможно считать сообщение из очереди, не изменив содержимое файла. Аналогично невозможно записать сообщение в очередь, не имея доступа на чтение. Обойти эту проблему можно, сохранив режим открытия (O_RDONLY, O_WRONLY, O_RDWR) в структуре mq_info и проверяя этот режим в каждой из функций. Например, mq_receive должна возвращать ошибку, если в mq_info хранится значение O_WRONLY.

116-132 Нам необходимо дождаться, когда очередь будет проинициализирована (в случае, если несколько потоков сделают попытку открыть ее приблизительно одновременно). Для этого мы вызываем stat и проверяем разрешения доступа к файлу (поле st_mode структуры stat). Если бит user-execute сброшен, очередь уже проинициализирована.

Этот участок кода обрабатывает другую возможную ситуацию гонок. Предположим, что два потока разных процессов пытаются открыть очередь приблизительно одновременно. Первый поток создает файл и блокируется при вызове lseek в листинге 5.18. Второй поток обнаруживает, что файл уже существует, и переходит к метке exists, где он вновь открывает файл функцией open и при этом блокируется. Затем продолжается выполнение первого потока, но его вызов mmap в листинге 5.18 не срабатывает (возможно, он превысил лимит использования памяти), поэтому он переходит на метку egg и удаляет созданный файл вызовом unlink. Продолжается выполнение второго потока, но если бы мы вызывали fstat вместо stat, он бы вышел по тайм-ауту в цикле for, ожидая инициализации файла. Вместо этого мы вызываем stat, которая возвращает ошибку, если файл не существует, и, если флаг O_CREAT был указан при вызове mq_open, мы переходим на метку again (листинг 5.17) для повторного создания файла. Эта ситуация гонок заставляет нас также проверять, не возвращается ли при вызове open ошибка ENOENT.

133-144 Файл отображается в память, после чего его дескриптор может быть закрыт. Затем мы выделяем место под структуру mq_info и инициализируем ее. Возвращаемое значение представляет собой указатель на эту структуру.

145-148 При возникновении ошибок происходит переход к метке egg, а переменной eggno присваивается значение, которое должно быть возвращено функцией mq_open. Мы аккуратно вызываем функции для очистки памяти от выделенных объектов, чтобы переменная eggno не изменила свое значение в случае возникновения ошибки в этих функциях.

Функция mq_close

В листинге 5.20 приведен текст нашей функции mq_close.

10-16 Проверяется правильность переданных аргументов, после чего получаются указатели на область, занятую отображенными в память файлом (mqhd़), и атрибуты (в структуре mq_hdr).

17-18 Для сброса регистрации на уведомление вызвавшего процесса мы вызываем mq_notify. Если процесс был зарегистрирован, он будет снят с уведомления, но если нет — ошибка не возвращается.

19-25 Мы вычисляем размер файла для вызова munmap и освобождаем память, используемую структурой mqinfo. На случай, если вызвавший процесс будет продолжать использовать дескриптор очереди сообщений, до того как область памяти будет вновь задействована вызовом malloc, мы устанавливаем значение mq_magic в ноль, чтобы наши функции для работы с очередью сообщений обнаруживали ошибку.

Обратите внимание, что если процесс завершает работу без вызова mq_close, эти же операции выполняются автоматически: отключается отображение в память, а память освобождается.

Функция mq_unlink

Текст функции mqunlink приведен в листинге 5.21. Она удаляет файл, связанный с очередью сообщений, вызывая функцию unlink.

Функция mq_getattr

В листинге 5.22 приведен текст функции mq_getattr, которая возвращает текущее значение атрибутов очереди.

17-20 Мы должны заблокировать соответствующее взаимное исключение для работы с очередью, в частности для получения атрибутов, поскольку какой-либо другой поток может в это время их изменить.

Функция mq_setattr

В листинге 5.23 приведен текст функции `mq_setattr`, которая устанавливает значение атрибутов очереди.

22-27 Если третий аргумент представляет собой ненулевой указатель, мы возвращаем предыдущее значение атрибутов перед внесением каких-либо изменений.

28-31 Единственный атрибут, который можно менять с помощью нашей функции, — `mq_flags`, хранящийся в структуре `mq_info`.

Функция mq_notify

Функция mq_notify, текст которой приведен в листинге 5.24, позволяет регистрировать процесс на уведомление для текущей очереди и снимать его с регистрации. Информация о зарегистрированных процессах (их идентификаторы) хранится в поле mqh_pid структуры mq_hdr. Только один процесс может быть зарегистрирован на уведомление в любой момент времени. При регистрации процесса мы сохраняем его структуру sigevent в структуре mqh_event.

20-24 Если второй аргумент представляет собой нулевой указатель, вызвавший процесс снимается с регистрации. Если он не зарегистрирован, никакой ошибки не возвращается.

25-34 Если какой-либо процесс уже зарегистрирован, мы проверяем, существует ли он, отправкой ему сигнала с кодом 0 (называемого нулевым сигналом — null signal). Это обычная проверка на возможность ошибки, на самом деле при этом никакого сигнала процессу не отправляется, но при его отсутствии возвращается ошибка с кодом ESRCH. Если какой-либо процесс уже зарегистрирован на уведомление, функция возвращает ошибку EBUSY. В противном случае сохраняется идентификатор процесса вместе с его структурой sigevent.

ПРИМЕЧАНИЕ

Наш метод проверки существования вызвавшего процесса не идеален. Процесс мог завершить работу, а его идентификатор мог быть использован другим процессом.

Функция mq_send

В листинге 5.25 приведен текст первой половины нашей функции mqsend.

14-29 Мы получаем указатели на используемые структуры и блокируем взаимное исключение для данной очереди. Проверяем, не превышает ли размер сообщения максимально допустимый для данной очереди.

30-38 Если мы помещаем первое сообщение в пустую очередь, нужно проверить, не зарегистрирован ли какой-нибудь процесс на уведомление об этом событии и нет ли потоков, заблокированных в вызове mq_receive. Для проверки второго условия мы воспользуемся сохраняемым функцией mq_receive счетчиком mqh_nwait, содержащим количество потоков, заблокированных в вызове mq_receive. Если этот счетчик имеет ненулевое значение, мы не отправляем уведомление зарегистрированному процессу. Для отправки сигнала SIGEV_SIGNAL используется функция sigqueue. Затем процесс снимается с регистрации.

ПРИМЕЧАНИЕ

Вызов sigqueue для отправки сигнала приводит к передаче сигнала SI_QUEUE обработчику сигнала в структуре типа `siginfo_t` (раздел 5.7), что неправильно. Отправка правильного значения `si_code` (а именно `SI_MESSAGE`) из пользовательского процесса осуществляется в зависимости от реализации. На с. 433 стандарта IEEE 1996 [8] отмечается, что для отправки этого сигнала из пользовательской библиотеки необходимо воспользоваться скрытым интерфейсом отправки сигналов.

39-48 Если очередь переполнена и установлен флаг `O_NONBLOCK`, мы возвращаем ошибку с кодом `EAGAIN`. В противном случае мы ожидаем сигнала по условной переменной `mqh_wait`, который, как мы увидим, отправляется функцией `mq_receive` при считывании сообщения из переполненной очереди.

ПРИМЕЧАНИЕ

Наша реализация упрощает ситуацию с возвращением ошибки `EINTR` при прерывании вызова `mq_send` сигналом, перехватываемым вызвавшим процессом. Проблема в том, что функция `pthread_cond_wait` не возвращает ошибки при возврате из обработчика сигнала: она может вернуть либо 0 (что рассматривается как ложное пробуждение), либо вообще не завершить работу. Все эти проблемы можно обойти, но это непросто.

В листинге 5.26 приведена вторая половина функции `mq_send`. К моменту ее выполнения мы уже знаем о наличии в очереди свободного места для нашего сообщения.

50-52 Поскольку количество свободных сообщений при создании очереди равно `mq_maxmsg`, ситуация, в которой `mq_curmsgs` будет меньше `mq_maxmsg` для пустого списка свободных сообщений, возникнуть не может.

53-56 Указатель `nmsghdr` хранит адрес области памяти, в которую помещается сообщение. Приоритет и длина сообщения сохраняются в структуре `msg_hdr`, а затем в память копируется содержимое сообщения, переданного вызвавшим процессом.

57-74 Порядок сообщений в нашем списке зависит от их приоритета: они расположены в порядке его убывания. При добавлении нового сообщения мы проверяем, существуют ли сообщения с тем же приоритетом; в этом случае сообщение добавляется после последнего из них. Используя такой метод упорядочения, мы гарантируем, что `mq_receive` всегда будет возвращать старейшее сообщение с наивысшим приоритетом. По мере продвижения по списку мы сохраняем в `pmsghdr` адрес предыдущего сообщения, поскольку именно это сообщение будет хранить индекс нового сообщения в поле `msg_next`.

ПРИМЕЧАНИЕ

Наша схема может оказаться медленной в случае наличия в очереди большого количества сообщений, поскольку каждый раз при добавлении нового придется просматривать их значительную часть. Можно хранить отдельно индексы последних сообщений со всеми имеющимися значениями приоритета.

75-77 Если очередь была пуста в момент помещения в нее нового сообщения, мы вызываем `pthread_cond_signal`, чтобы разблокировать любой из процессов, ожидающих сообщения.

78 Увеличиваем на единицу количество сообщений в очереди `mq_curmsgs`.

Функция mq_receive

В листинге 5.27 приведен текст первой половины функции mq_receive, которая получает необходимые указатели, блокирует взаимное исключение и проверяет объем буфера вызвавшего процесса, который должен быть достаточным для помещения туда сообщения максимально возможной длины.

30-40 Если очередь пуста и установлен флаг O_NONBLOCK, возвращается ошибка с кодом EAGAIN. В противном случае увеличивается значение счетчика mqh_nwait, который проверяется функцией mq_send (листинг 5.25) в случае, если очередь пуста и есть процессы, ожидающие уведомления. Затем мы ожидаем сигнала по условной переменной, который будет передан функцией mq_send (листинг 5.26).

ПРИМЕЧАНИЕ

Наша реализация mq_receive, как и реализация mq_send, упрощает ситуацию с ошибкой EINTR, возвращаемой при прерывании ожидания сигналом, перехватываемым вызвавшим процессом.

В листинге 5.28 приведен текст второй половины функции mq_receive. Мы уже знаем, что в очереди есть сообщение, которое можно будет возвратить вызвавшему процессу.

43-51 msghdr указывает на msg_hdr первого сообщения в очереди, которое мы и возвратим. Освободившееся сообщение становится первым в списке свободных.

52-54 Если очередь была полной в момент считывания сообщения, мы вызываем pthread_cond_signal для отправки сообщения любому из процессов, заблокированных в вызове mq_send.

5.9. Резюме

Очереди сообщений Posix просты в использовании: новая очередь создается (или существующая открывается) функцией `mq_open`; закрываются очереди вызовом `mq_close`, а удаляются `mq_unlink`. Поместить сообщение в очередь можно функцией `mq_send`, а считать его оттуда можно с помощью `mq_receive`. Атрибуты очереди можно считать и установить с помощью функций `mq_getattr` и `mq_setattr`, а функция `mq_notify` позволяет зарегистрировать процесс на уведомление о помещении нового сообщения в пустую очередь. Каждое сообщение в очереди обладает приоритетом (небольшое целое число), и функция `mq_receive` всегда возвращает старейшее сообщение с наивысшим приоритетом.

Изучая `mq_notify`, мы познакомились с сигналами реального времени стандарта Posix, которые обладают номерами от `SIGMIN` до `SIGMAX`. При установке обработчика для этих сигналов с флагом `SA_SIGINFO` они будут помещаться в очередь, доставляться в порядке очереди и сопровождаться двумя дополнительными аргументами (при вызове обработчика).

Наконец, мы реализовали большую часть возможностей очереди сообщений Posix в приблизительно 500 строках кода на языке C, используя отображаемые в память файлы, взаимные исключения и условные переменные Posix. Эта реализация иллюстрирует обработку ситуации гонок при создании новой очереди; еще раз нам придется столкнуться с такой ситуацией в главе 10 при реализации семафоров Posix.

Упражнения

1. Говоря о листинге 5.4, мы отметили, что атрибут *attr* функции *mq_open* при создании новой очереди является ненулевым; следует указать оба поля: *mq_maxmsg* и *mq_msgsize*. Как можно было бы указать только одно из них, не указывая второе, для которого использовать значения атрибутов по умолчанию?
2. Измените листинг 5.8 так, чтобы при получении сигнала не вызывалась функция *mq_notify*. Затем поместите в очередь два сообщения и убедитесь, что для второго из них сигнал порожден не будет. Почему?
3. Измените листинг 5.8 так, чтобы сообщение из очереди при получении сигнала не считывалось. Вместо этого просто вызовите *mq_notify* и напечатайте сообщение о получении сигнала. Затем отправьте два сообщения и убедитесь, что для второго из них сигнал не порождается. Почему?
4. Что произойдет, если мы уберем преобразование двух констант к целому типу в первом вызове *printf* в листинге 5.14?
5. Измените листинг 5.4 следующим образом: перед вызовом *mq_open* напечатайте сообщение и подождите 30 секунд (*sleep*). После возвращения из *mq_open* выведите еще одно сообщение и подождите еще 30 секунд, а затем вызовите *mq_close*. Откомпилируйте программу и запустите ее, указав большое количество сообщений (несколько сотен тысяч) и максимальный размер сообщения, скажем, в 10 байт. Задача заключается в том, чтобы создать большую очередь и проверить, используются ли в реализации отображаемые в память файлы. В течение 30-секундной паузы запустите программу типа *ps* и посмотрите на занимаемый программой объем памяти. Сделайте это еще раз после возвращения из *mq_open*. Можете ли вы объяснить происходящее?
6. Что произойдет при вызове *memscrub* в листинге 5.26, если вызвавший процесс укажет нулевую длину сообщения?
7. Сравните очередь сообщений с двусторонними каналами, описанными в разделе 4.4. Сколько очередей нужно для двусторонней связи между родительским и дочерним процессами?
8. Почему мы не удаляем взаимное исключение и условную переменную в листинге 5.20?
9. Стандарт Posix утверждает, что дескриптор очереди сообщений не может иметь тип массива. Почему?
10. В каком состоянии проводит большую часть времени функция *main* из листинга 5.12? Что происходит каждый раз при получении сигнала? Как мы обрабатываем эту ситуацию?
11. Не все реализации поддерживают атрибут *PTHREAD_PROCESS_SHARED* для взаимных исключений и условных переменных. Переделайте реализацию очередей сообщений из раздела 5.8 так, чтобы использовать семафоры Posix (глава 10) вместо взаимных исключений и условных переменных.
12. Расширьте реализацию очередей сообщений Posix из раздела 5.8 так, чтобы она поддерживала *SIGEV_THREAD*.

6.1. Введениеы

Каждой очереди сообщений System V сопоставляется свой *идентификатор очереди сообщений*. Любой процесс с соответствующими привилегиями (раздел 3.5) может поместить сообщение в очередь, и любой процесс с другими соответствующими привилегиями может сообщение из очереди считать. Как и для очередей сообщений Posix, для помещения сообщения в очередь System V не требуется наличия подключенного к ней на считывание процесса.

Ядро хранит информацию о каждой очереди сообщений в виде структуры, определенной в заголовочном файле `<sys/msg.h>`:

ПРИМЕЧАНИЕ

Unix 98 не требует наличия полей `msg_first`, `msg_last` и `msg_cbytes`. Тем не менее они имеются в большинстве существующих реализаций, производных от System V. Естественно, ничто не заставляет реализовывать очередь сообщений через связный список, который неявно предполагается при наличии полей `msg_first` и `msg_last`. Эти два указателя обычно указывают на участки памяти, принадлежащие ядру, и практически бесполезны для приложения.

Мы можем изобразить конкретную очередь сообщений, хранимую ядром как связный список, — рис. 6.1. В этой очереди три сообщения длиной 1, 2 и 3 байта с типами 100, 200 и 300 соответственно.

В этой главе мы рассмотрим функции, используемые для работы с очередями сообщений System V, и реализуем наш пример файлового сервера из раздела 4.2 с использованием очередей сообщений.



Рис. 6.1. Структура очереди system V в ядре

6.2. Функция msgget

Создать новую очередь сообщений или получить доступ к существующей можно с помощью функции msgget:

Возвращаемое значение представляет собой целочисленный идентификатор, используемый тремя другими функциями msg для обращения к данной очереди. Идентификатор вычисляется на основе указанного *ключа*, который может быть получен с помощью функции ftok или может представлять собой константу IPC_PRIVATE, как показано на рис. 3.1.

Флаг *oflag* представляет собой комбинацию разрешений чтения-записи, показанную в табл. 3.3. К разрешениям можно добавить флаги IPC_CREAT или IPC_CREAT | IPC_EXCL с помощью логического сложения, как уже говорилось в связи с рис. 3.2.

При создании новой очереди сообщений инициализируются следующие поля структуры msqid_ds:

- полям uid и cuid структуры msg_perm присваивается значение действующего идентификатора пользователя вызвавшего процесса, а полям gid и cgid — действующего идентификатора группы;
- разрешения чтения-записи, указанные в oflag, помещаются в msg_perm.mode;
- значения msg_qnum, msg_lspid, msg_lrpid, msg_stime и msg_rtime устанавливаются в 0;
- в msg_ctime записывается текущее время;
- в msg_qbytes помещается системное ограничение на размер очереди.

6.3. Функция msgsnd

После открытия очереди сообщений с помощью функции msgget можно помещать сообщения в эту очередь с помощью msgsnd.

Здесь *msqid* представляет собой идентификатор очереди, возвращаемый msgget. Указатель *ptr* указывает на структуру следующего шаблона, определенного в <sys/ msg.h>:

Тип сообщения должен быть больше нуля, поскольку неположительные типы используются в качестве специальной команды функции msgrcv, о чем рассказывается в следующем разделе.

Название *intext* в структуре *msgbuf* употреблено не вполне правильно; данные в сообщении совсем не обязательно должны быть текстом. Разрешена передача любых типов данных как в двоичном, так и в текстовом формате. Ядро никак не интерпретирует содержимое сообщения.

Для описания структуры мы используем термин «шаблон», поскольку *ptr* указывает на целое типа long, представляющее собой тип сообщения, за которым непосредственно следует само сообщение (если его длина больше 0 байт). Большинство приложений не пользуются этим определением структуры *msgbuf*, поскольку установленного в ней количества данных (1 байт) обычно недостаточно для прикладных задач. На количество данных в сообщении никаких ограничений при компиляции не накладывается (как правило, оно может быть изменено системным администратором), поэтому вместо объявления структуры с большим объемом данных (большим, чем поддерживается текущей реализацией) определяется этот шаблон. Большинство приложений затем определяют собственную структуру сообщений, в которой передаваемые данные зависят от нужд этих приложений.

Например, если приложению нужно передавать сообщения, состоящие из 16-разрядного целого, за которым следует 8-байтовый массив символов, оно может определить свою собственную структуру так:

Аргумент *length* функции msgsnd указывает длину сообщения в байтах. Это длина пользовательских данных, следующих за типом сообщения (целое типа long). Длина может быть и 0. В указанном выше примере длина может быть вычислена как sizeof(Message) - sizeof(long).

Аргумент *flag* может быть либо 0, либо IPC_NOWAIT. В последнем случае он отключает блокировку для msgsnd: если для нового сообщения недостаточно места в очереди, возврат из функции происходит немедленно. Это может произойти, если:

- в данной очереди уже имеется слишком много данных (значение *msg_qbytes* в структуре *msqid_ds*);
- во всей системе имеется слишком много сообщений.

Если верно одно из этих условий и установлен флаг IPC_NOWAIT, функция msgsnd возвращает ошибку с кодом EAGAIN. Если флаг IPC_NOWAIT не указан, а одно из этих условий выполняется, поток приостанавливается до тех пор, пока не произойдет одно из следующего:

- для сообщения освободится достаточно места;
- очередь с идентификатором *msqid* будет удалена из системы (в этом случае возвращается ошибка с кодом EIDRM);
- вызвавший функцию поток будет прерван перехватываемым сигналом (в этом случае возвращается ошибка с кодом EINTR).

6.4. Функция msgrcv

Сообщение может быть считано из очереди с помощью функции msgrcv.

Аргумент *ptr* указывает, куда следует помещать принимаемые данные. Как и для msgsnd, он указывает на поле данных типа long (рис. 4.13), которое непосредственно предшествует полезным данным.

Аргумент *length* задает размер относящейся к полезным данным части буфера, на который указывает *ptr*. Это максимальное количество данных, которое может быть возвращено функцией. Поле типа long не входит в эту длину.

Аргумент *type* определяет тип сообщения, которое нужно считать из очереди:

- если значение *type* равно 0, возвращается первое сообщение в очереди (то есть при указании типа 0 возвращается старейшее сообщение);
- если тип больше 0, возвращается первое сообщение, тип которого равен указанному;
- если тип меньше нуля, возвращается первое сообщение с наименьшим типом, значение которого меньше либо равно модулю аргумента *type*.

Рассмотрим пример очереди сообщений, изображенный на рис. 6.1. В этой очереди имеются три сообщения:

- первое сообщение имеет тип 100 и длину 1;
- второе сообщение имеет тип 200 и длину 2;
- третье сообщение имеет тип 300 и длину 3.

Таблица 6.1 показывает, какое сообщение будет возвращено при различных значениях аргумента *type*.

Таблица 6.1. Возвращаемое сообщение в зависимости от аргумента *type*

Аргумент *flag* указывает, что делать, если в очереди нет сообщения с запрошенным типом. Если установлен бит IPC_NOWAIT, происходит немедленный возврат из функции msgrcv с кодом ошибки ENOMSG. В противном случае вызвавший процесс блокируется до тех пор, пока не произойдет одно из следующего:

- появится сообщение с запрошенным типом;
- очередь с идентификатором *msqid* будет удалена из системы (в этом случае будет возвращена ошибка с кодом EIDRM);
- вызвавший поток будет прерван перехватываемым сигналом (в этом случае возвращается ошибка EINTR).

В аргументе *flag* можно указать дополнительный бит MSG_NOERROR. При установке этого бита данные, превышающие объем буфера (аргумент *length*), будут просто обрезаться до его размера без возвращения кода ошибки. Если этот флаг не указать, при превышении объемом сообщения аргумента *length* будет возвращена ошибка E2BIG.

В случае успешного завершения работы msgrcv возвращает количество байтов в принятом сообщении. Оно не включает байты, нужные для хранения типа сообщения (long), который также возвращается через указатель *ptr*.

Функция `msgctl` позволяет управлять очередями сообщений:

Команд (аргумент *cmd*) может быть три:

- `IPC_RMID` — удаление очереди с идентификатором *msqid* из системы. Все сообщения, имеющиеся в этой очереди, будут потеряны. Мы уже видели пример действия этой функции в листинге 3.2. Для этой команды третий аргумент функции игнорируется.
- `IPC_SET` — установка значений четырех полей структуры `msqid_ds` данной очереди равными значениям соответствующих полей структуры, на которую указывает аргумент *buff*: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, `msg_qbytes`.
- `IPC_STAT` — возвращает вызвавшему процессу (через аргумент *buff*) текущее содержимое структуры `msqid_ds` для указанной очереди.

Пример

Программа в листинге 6.1 создает очередь сообщений, помещает в нее сообщение с 1 байтом информации, вызывает функцию msgctl с командой IPC_STAT, выполняет команду ipcs, используя функцию system, а затем удаляет очередь, вызвав функцию msgctl с командой IPC_RMID.

Мы собираемся отправить сообщение размером 1 байт, поэтому можно просто воспользоваться стандартным определением структуры msgbuf из <sys/msg.h>. Выполнение этой программы приведет к следующему результату:

Выведенные значения соответствуют ожидаемым. Нулевое значение ключа обычно соответствует IPC_PRIVATE, как мы отмечали в разделе 3.2. В этой системе на очередь сообщений накладывается ограничение по объему в 4096 байт. Поскольку мы записали сообщение с 1 байтом данных и msg_cbytes имеет значение 1, это ограничение накладывается на объем полезных данных и не включает тип сообщения (long), указываемый для каждого сообщения.

Поскольку очереди сообщений System V обладают живучестью ядра, мы можем написать несколько отдельных программ для работы с этими очередями и изучить их действие.

Программа msgcreate

В листинге 6.2 приведена программа msgcreate, создающая очередь сообщений.

9-12 Параметр командной строки -e позволяет указать флаг IPC_EXCL.

16 Полное имя файла, являющееся обязательным аргументом командной строки, передается функции ftok. Получаемый ключ преобразуется в идентификатор функцией msgget.

Программа msgsnd

Программа msgsnd приведена в листинге 6.3. Она помещает в очередь одно сообщение заданной длины и типа.

Мы создаем указатель на структуру msgbuf общего вида, а затем выделяем место под реальную структуру (буфер записи) соответствующего размера, вызвав calloc. Эта функция инициализирует буфер нулем.

Программа msgrcv

В листинге 6.4 приведен текст программы msgrcv,читывающей сообщение из очереди. В командной строке может быть указан параметр -n, отключающий блокировку, а параметр -t может быть использован для указания типа сообщения в функции msgrcv.

2 Не существует простого способа определить максимальный размер сообщения (об этом и других ограничениях мы поговорим в разделе 6.10), поэтому мы определим свою собственную константу.

Программа msgrmid

Для удаления очереди сообщений мы вызываем функцию msgctl с командой IPC_RMID, как показано в листинге 6.5.

Примеры

Теперь воспользуемся четырьмя только что написанными программами. Создадим очередь и поместим в нее три сообщения:

Сначала мы пытаемся создать очередь, используя имя несуществующего файла. Пример показывает, что файл, указываемый в качестве аргумента ftok, обязательно должен существовать. Затем мы создаем файл /tmp/test1 и используем его имя при создании очереди сообщений. После этого в очередь помещаются три сообщения длиной 1, 2 и 3 байта со значениями типа 100, 200 и 300 (вспомните рис. 6.1). Программа ipcs показывает, что в очереди находятся 3 сообщения общим объемом 6 байт.

Теперь продемонстрируем использование аргумента type при вызове msgrcv для считывания сообщений в произвольном порядке:

В первом примере запрашивается сообщение с типом 200, во втором примере — сообщение с наименьшим значением типа, не превышающим 300, а в третьем — первое сообщение в очереди. Последний запуск msgrcv иллюстрирует действие флага IPC_NOWAIT.

Что произойдет, если мы укажем положительное значение типа, а сообщений с таким типом в очереди не обнаружится?

Сначала мы вызываем ipcs, чтобы убедиться, что очередь пуста, а затем помещаем в нее сообщение длиной 1 байт с типом 100. Затем мы запрашиваем сообщение с типом 999, и программа блокируется (при вызове msgrcv), ожидая помещения в очередь сообщения с указанным типом. Мы прерываем ожидание нажатием клавиши. Затем мы запускаем программу с флагом -n, предотвращающим блокировку, и видим, что в этом случае возвращается ошибка с кодом ENOMSG. После этого мы удаляем очередь с помощью программы msgrmid. Мы могли бы удалить очередь и с помощью системной команды

в которой указывается идентификатор очереди, или с помощью той же команды в другом формате где указывается ключ очереди сообщений.

Программа msgrcvid

Покажем теперь, что для получения доступа к очереди сообщений System V не обязательно вызывать msgget: все, что нужно, — это знать идентификатор очереди сообщений, который легко получить с помощью ipcs, и считать разрешения доступа для очереди. В листинге 6.6 приведен упрощенный вариант программы msgrcv из листинга 6.4.

Здесь мы уже не используем msgget. Вместо этого используется идентификатор очереди сообщений, являющийся обязательным аргументом командной строки.

Вот пример использования этой программы:

Идентификатор очереди (150) мы узнали с помощью ipcs, его мы и предоставляем программе msgrcvid в качестве аргумента командной строки.

Этот же метод можно использовать для семафоров System V (упражнение 11.1) и разделяемой памяти System V (упражнение 14.1).

6.7. Пример программы клиент-сервер

Перепишем наш пример программы типа клиент-сервер из раздела 4.2 с использованием двух очередей сообщений. Одна из очередей предназначена для передачи сообщений от клиента серверу, а другая — в обратную сторону.

Заголовочный файл svmsg.h приведен в листинге 6.7. Мы подключаем наш стандартный заголовочный файл и определяем ключи для каждой из очередей сообщений.

Функция main для сервера приведена в листинге 6.8. Программа создает обе очереди сообщений, и не беда, если какая-нибудь из них уже существует, потому что мы не указываем флаг IPC_EXCL. Функция server дана в листинге 4.16. Она вызывает наши собственные функции mesgsend и mesgrecv, новые версии которых будут приведены ниже.

В листинге 6.9 приведен текст функции main программы-клиента. Программа открывает две очереди сообщений и вызывает функцию client из листинга 4.15. Эта функция использует две другие: mesg_send и mesg_recv, которые будут приведены ниже.

И функция client, и функция server используют формат сообщений, изображенный в листинге 4.12. Для передачи и приема сообщений они используют функции mesg_send и mesg_recv. Старые версии этих функций, приведенные в листингах 4.13 и 4.14, вызывали write и read и работали с программными каналами и FIFO, так что нам придется переписать их для использования очередей сообщений. В листингах 6.10 и 6.11 приведены новые версии этих функций. Обратите внимание, что аргументы функций не изменились, поскольку первый целочисленный аргумент может содержать как целочисленный дескриптор программного канала или FIFO, так и целочисленный дескриптор очереди сообщений.

Наличие поля type у каждого сообщения в очереди предоставляет две интересные возможности:

1. Поле type может использоваться для идентификации сообщений, позволяя нескольким процессам мультиплексировать сообщения в одной очереди. Например, все сообщения от клиентов серверу имеют одно и то же значение типа, тогда как сообщения сервера клиентам имеют различные значения типов, уникальные для каждого клиента. Естественно, в качестве значения типа сообщения, гарантированно уникального для каждого клиента, можно использовать идентификатор процесса клиента.
2. Поле type может использоваться для установки приоритета сообщений. Это позволяет получателю считывать сообщения в порядке, отличном от обычного для очередей (FIFO). В программных каналах и FIFO данные могли приниматься только в том порядке, в котором они были отправлены. Очереди System V позволяют считывать сообщения в произвольном порядке в зависимости от значений типа сообщений. Более того, можно вызывать msgrcv с флагом IPC_NOWAIT для считывания сообщений с конкретным типом и немедленного возвращения управления процессу в случае отсутствия таких сообщений.

Пример: одна очередь на приложение

Вспомните наш простой пример с одним процессом-сервером и одним процессом-клиентом. Если применять программные каналы или FIFO, необходимо наличие двух каналов IPC для передачи данных в обоих направлениях, поскольку эти типы IPC являются однонаправленными. Очереди сообщений позволяют передавать данные в обоих направлениях, причем поле *type* может использоваться для указания адресата (клиента или сервера).



Рис. 6.2. Мультиплексирование сообщений между несколькими клиентами и одним сервером

Рассмотрим усложненный вариант: один сервер и несколько клиентов. В этом случае можно использовать значение типа 1, например, для обозначения сообщений от любого клиента серверу. Если клиент передаст серверу свой идентификатор процесса в качестве части сообщения, сервер сможет отсылать клиенту сообщения, используя его идентификатор в качестве значения типа сообщения. Каждый клиент будет использовать свой PID в качестве аргумента *type* при вызове `msgrecv`. На рис. 6.2 приведен пример использования очереди для мультиплексирования этих сообщений между несколькими клиентами и одним сервером.

ПРИМЕЧАНИЕ

При использовании одного канала IPC одновременно клиентами и сервером всегда существует потенциальная возможность зависания (deadlock). Клиенты могут (в этом примере) заполнить очередь своими сообщениями, не давая серверу возможности отправить ответ. В этом случае клиенты заблокируются при вызове `msgsnd`, как и сервер. Одно из соглашений, исключающих возможность такой взаимной блокировки, заключается в том, что сервер должен всегда отключать блокировку записи в очередь сообщений.

Теперь мы можем переделать наш пример с клиентом и сервером, используя одну очередь сообщений с различными типами для разных адресатов. Эти программы используют следующее соглашение: сообщения с типом 1 адресованы серверу, а все остальные сообщения имеют тип, соответствующий идентификатору процесса адресата. При этом запрос клиента должен содержать его PID вместе с полным именем запрашиваемого файла, аналогично программе в разделе 4.8.

В листинге 6.12 приведен текст функции `main` сервера. Заголовочный файл `svmsg.h` был приведен в листинге 6.7. Создается единственная очередь сообщений (если она существует, ошибки не возникнет). Идентификатор этой очереди сообщений используется в качестве обоих аргументов при вызове функции `server`.

Функция `server` обеспечивает работу сервера. Ее текст приведен в листинге 6.13. Эта функция представляет собой комбинацию листинга 4.10 — нашего сервера FIFO, считывавшего команды, состоявшие из идентификатора процесса и полного имени файла, — и листинга 4.16, в котором использовались функции `mesg_send` и `mesg_recv`. Обратите внимание, что идентификатор процесса, отправляемый клиентом, используется в качестве типа для всех сообщений, отправляемых сервером этому клиенту. Эта функция представляет собой бесконечный цикл, в которомчитываются запросы клиентов и отсылаются запрошенные файлы. Этот сервер является последовательным (см. раздел 4.9).

В листинге 6.14 приведен текст функции `main` клиента. Клиент открывает очередь сообщений, которая должна была быть создана сервером заранее.

Функция `client`, текст которой дан в листинге 6.15, обеспечивает всю обработку со стороны клиента. Эта функция представляет собой комбинацию программ из листингов 4.11 и 4.15. В первой программе клиент отсылал свой идентификатор и полное имя файла, а во второй программе использовались функции `mesg_send` и `mesg_recv`. Обратите внимание, что тип сообщений, запрашиваемых функцией `mesg_recv`, совпадает с идентификатором процесса клиента.

Функции `client` и `server` используют функции `mesg_send` и `mesg_recv` из листингов 6.9 и 6.11.

Пример: одна очередь для каждого клиента

Изменим теперь предыдущий пример таким образом, чтобы все запросы клиентов передавались по одной очереди, но для отправки ответов использовалась бы отдельная очередь для каждого клиента. На рис. 6.3 изображена схема такого приложения.



Рис. 6.3. Одна очередь для сервера и по одной для каждого клиента

Ключ очереди сервера должен быть известен клиентам, а сами клиенты создают свои очереди с ключом IPC_PRIVATE. Вместо передачи серверу идентификатора процесса клиенты сообщают ему идентификатор своей очереди, в которую сервер направляет свой ответ. Этот сервер является параллельным: для каждого нового клиента порождается отдельный процесс.

ПРИМЕЧАНИЕ

При такой схеме может возникнуть проблема в случае «гибели» клиента, потому что тогда сообщения останутся в его очереди навсегда (по крайней мере до перезагрузки ядра или явного удаления очереди другим процессом).

Нижеследующие заголовочные файлы и функции не претерпевают изменений по сравнению с предыдущими версиями:

- mesg.h (листинг 4.12);
- svmsg.h (листинг 6.7);
- функция main сервера (листинг 6.12);
- функция mesg_send (листинг 4.13).

Функция main клиента приведена в листинге 6.16; она слегка изменилась по сравнению с листингом 6.14. Мы открываем очередь сервера с известным ключом (MQ_KEY1) и создаем нашу собственную очередь с ключом IPC_PRIVATE. Два идентификатора этих очередей становятся аргументами функции client (листинг 6.17). После завершения работы клиента его персональная очередь удаляется.

В листинге 6.17 приведен текст функции client. Эта функция практически идентична функции из листинга 6.15, но вместо передачи идентификатора процесса клиента на сервер направляется идентификатор очереди клиента. Тип сообщения в структуре mesg остается равным 1, поскольку это значение устанавливается для сообщений, передаваемых в обе стороны.

В листинге 6.19 приведена функция server. Главное отличие от листинга 6.13 в том, что эта функция представляет собой бесконечный цикл, в котором для каждого нового клиента вызывается fork.

10 Поскольку для каждого клиента порождается отдельный процесс, нужно позаботиться о процессах-зомби. В разделах 5.9 и 5.10 [24] об этом говорится подробно. Здесь мы просто установим обработчик для сигнала SIGCHLD, и наша функция sig_chld (листинг 6.18) будет вызываться при завершении работы дочернего процесса.

12-18 Породивший процесс сервера блокируется в вызове mesg_recv, ожидая появления сообщения от очередного клиента.

25-45 Вызовом fork порождается новый процесс, который производит попытку открыть запрошенный файл и отправляет клиенту либо сообщение об ошибке, либо содержимое файла. Мы преднамеренно поместили вызов fopen в дочерний процесс, а не в родительский, поскольку если файл находится в удаленной файловой системе, его открытие может занять довольно много времени в случае наличия проблем с сетью.

Функция-обработчик для SIGCHLD приведена в листинге 6.18. Она скопирована с листинга 5.11 [24].

Каждый раз при вызове обработчика происходит циклический вызов `waitpid` для получения статуса завершения работы всех дочерних процессов, которые могли завершить работу. Затем происходит возврат из обработчика сигнала. При этом может возникнуть проблема, поскольку родительский процесс проводит большую часть времени в заблокированном состоянии (при вызове `mesg_recv`, листинг 6.9). При возвращении из обработчика этот вызов `msgrecv` прерывается. Функция возвращает ошибку с кодом `EINTR`, как рассказывается в разделе 5.9 [24].

Нам нужно обработать такой возврат из вызванной функции, поэтому мы пишем новую функцию-обертку `Mesg_recv`, приведенную в листинге 6.20. Эта программа допускает возвращение ошибки с кодом `EINTR` функцией `mesg_recv` (которая просто вызывает `msgrecv`), и, если это происходит, мы просто еще раз вызываем `mesg_recv`.

6.9. Использование select и poll с очередями сообщений

Одним из недостатков очередей сообщений System V является то, что они идентифицируются не дескрипторами, а идентификаторами. Поэтому с ними нельзя использовать функции select и poll (глава 6 [24]).

ПРИМЕЧАНИЕ

На самом деле одна из версий Unix, а именно AIX (созданная IBM), позволяет использовать select с очередями сообщений System V, а не только с дескрипторами. Но эта возможность имеется только в AIX.

Этот недостаток часто всплывает, когда возникает необходимость написать сервер, работающий одновременно с сетевыми соединениями и с IPC. Сетевые соединения с использованием интерфейса сокетов или XTI ([24]) используют дескрипторы, что позволяет вызывать select или poll.

Программные каналы и FIFO также идентифицируются дескрипторами, поэтому для них тоже допустимо использование этих функций.

Одним из решений этой проблемы является следующее: сервер должен создать канал и породить процесс, который будет заблокирован при вызове msgrcv. При получении сообщения произойдет возврат из msgrcv, дочерний процесс получит это сообщение из очереди и запишет его в канал. Затем родительский процесс может использовать функцию select для канала совместно с сетевыми соединениями. Недостаток этого подхода в том, что сообщения обрабатываются трижды: при считывании дочерним процессом с помощью msgrcv, при отправке в канал и при считывании из канала родительским процессом. Для ускорения обработки порожденный процесс может создать сегмент совместно используемой с породившим процессом памяти, а канал использовать как флаг (упражнение 12.5).

ПРИМЕЧАНИЕ

В листинге 5.12 мы привели решение с использованием очередей сообщений Posix, которое не требовало вызова fork. Для очередей сообщений Posix можно было обойтись одним процессом, поскольку они предусматривают уведомление о появлении нового сообщения с помощью сигнала. Для очередей System V такая возможность не предусмотрена, поэтому приходится порождать процесс, который будет блокироваться при вызове msgrcv.

Другим недостатком очередей сообщений System V по сравнению с сетевым интерфейсом является невозможность считывания сообщений из оперативной памяти (возможность, предоставляемая флагом MSG_PEEK для функций grecv, recvfrom, recvmsg [24, с. 356]). Если бы такая возможность имелась, в предложенной только что схеме клиент-сервер (для обхода проблемы с select) можно было бы сделать работу более эффективной, указав флаг peek при вызове msgrcv дочерним процессом и записав 1 байт в канал при приходе сообщения, а родительский процесс тогда просто считывал бы сообщение из очереди.

Как отмечалось в разделе 3.8, на очереди сообщений часто накладываются системные ограничения. В табл. 6.2 приведены значения этих ограничений для двух конкретных реализаций. Первая колонка представляет собой традиционное имя System V для переменной ядра, хранящей это ограничение.

Таблица 6.2. Характерные значения ограничений для очередей сообщений

В этом разделе мы хотели показать типичные значения ограничений, чтобы помочь в планировании переносимых программ. При выполнении приложений, активно использующих очереди сообщений, обычно требуется настройка этих (или аналогичных) параметров ядра (что описано в разделе 3.8).

Пример

В листинге 6.21 приведен текст программы, которая определяет четыре ограничения, показанные в табл. 6.2.

14-28 Для определения максимально возможного размера сообщения мы пытаемся послать сообщение, в котором будет 65 536 байт данных, и если эта попытка оказывается неудачной, уменьшаем этот объем до 65 408, и т.д., пока вызов msgsnd не окажется успешным.

29-44 Теперь мы начинаем с 8-байтовых сообщений и смотрим, сколько их поместится в очередь. После определения этого ограничения мы удаляем очередь (сбрасывая все эти сообщения) и повторяем процедуру с 16-байтовыми сообщениями. Мы повторяем это до тех пор, пока не будет достигнут максимальный размер сообщения из первого пункта. Ожидается, что небольшие сообщения будут превышать ограничение по количеству сообщений в очереди, а большие — ограничение по количеству байтов.

45-54 Обычно есть системное ограничение на количество одновременно открытых идентификаторов. Оно определяется непосредственно созданием очередей до тех пор, пока не произойдет ошибка при вызове msgget.

Запустим эту программу сначала в Solaris 2.6, а затем в Digital Unix 4.0B, и результаты подтвердят приведенные в табл. 6.2 величины:

Причина, по которой в Digital Unix 4.0B получился результат 63 идентификатора, а не 64, как в табл. 6.2, заключается в том, что один идентификатор всегда используется системным демоном.

6.11.Резюме

Очереди сообщений System V аналогичны очередям сообщений Posix. При создании новых приложений следует рассмотреть возможность использования очередей сообщений Posix, но большое количество существующих программ использует очереди сообщений System V. Тем не менее переписать их для использования очередей Posix вместо очередей System V не должно быть слишком сложно. Главный недостаток очередей Posix заключается в невозможности считывания сообщений с произвольным приоритетом. Ни один из типов очередей не использует обычные дескрипторы, что делает сложным применение функций select и poll для очередей сообщений.

Упражнения

1. Почему на рис. 6.2 для сообщений, передаваемых серверу, используется тип 1?
2. Что произойдет с программой с рис. 6.2, если злоумышленник отправит на сервер множество сообщений, но не будет считывать ответы? Что в такой же ситуации произойдет с программой с рис. 6.3?
3. Переделайте реализацию очередей сообщений Posix из раздела 5.8 для использования очередей сообщений System V вместо отображения в память.

7.1. Введение

Эта глава начинается с обсуждения синхронизации — принципов синхронизации действий нескольких программных потоков или процессов. Обычно это требуется для предоставления нескольким потокам или процессам совместного доступа к данным. Взаимные исключения (mutual exclusion — mutex) и условные переменные (conditional variables) являются основными средствами синхронизации.

Взаимные исключения и условные переменные появились в стандарте Posix.1 для программных потоков, и всегда могут быть использованы для синхронизации отдельных потоков одного процесса. Стандарт Posix также разрешает использовать взаимное исключение или условную переменную и для синхронизации нескольких процессов, если это исключение или переменная хранится в области памяти, совместно используемой процессами.

ПРИМЕЧАНИЕ

Эта возможность является дополнительной согласно Posix, но обязательной в Unix 98 (см. табл. 1.3).

В этой главе мы разберем классическую схему производитель-потребитель, используя взаимные исключения и условные переменные. В примере будут использоваться программные потоки, а не процессы, поскольку предоставить потокам общий буфер данных, предполагаемый в этой задаче, легко, а вот создать буфер данных между процессами можно только с помощью одной из форм разделяемой памяти (которая будет описана только в четвертой части книги). Еще одно решение этой задачи, уже с использованием семафоров, появится в главе 10.

7.2. Взаимные исключения: установка и снятие блокировки

Взаимное исключение (mutex) является простейшей формой синхронизации. Оно используется для защиты критической области (critical region), предотвращая одновременное выполнение участка кода несколькими потоками (если взаимное исключение используется потоками) или процессами (если взаимное исключение используется несколькими процессами). Выглядит это обычно следующим образом:

Поскольку только один поток может заблокировать взаимное исключение в любой момент времени, это гарантирует, что только один поток будет выполнять код, относящийся к критической области.

Взаимные исключения по стандарту Posix объявлены как переменные с типом `pthread_mutex_t`. Если переменная-исключение выделяется статически, ее можно инициализировать константой `PTHREAD_MUTEX_INITIALIZER`:

При динамическом выделении памяти под взаимное исключение (например, вызовом `malloc`) или при помещении его в разделяемую память мы должны инициализировать эту переменную во время выполнения, вызвав функцию `pthread_mutex_init`, как показано в разделе 7.7.

ПРИМЕЧАНИЕ

Вам может попасться код, в котором взаимные исключения не инициализируются, поскольку в реализации инициализирующая константа имеет значение 0, а статические переменные автоматически инициализируются этим значением. Однако такой код некорректен.

Следующие три функции используются для установки и снятия блокировки взаимного исключения:

При попытке заблокировать взаимное исключение, которое уже заблокировано другим потоком, функция `pthread_mutex_lock` будет ожидать его разблокирования, а `pthread_mutex_trylock` (неблокируемая функция) вернет ошибку с кодом `EBUSY`.

ПРИМЕЧАНИЕ

Если несколько процессов ожидают освобождения взаимного исключения, какой из них начнет выполнятся первым? Одна из возможностей, добавленных стандартом 1003.1b-1993, заключается в установке приоритета потоков. Мы не будем углубляться в эту область, отметим лишь, что разным потокам могут быть присвоены разные значения приоритета и функции синхронизации (работающие с взаимными исключениями, блокировками чтения-записи и семафорами) будут передавать управление заблокированному потоку с наивысшим значением приоритета. Раздел 5.5 книги [3] описывает возможности планирования выполнения в Posix.1 более подробно.

Хотя мы говорим о защите критической области кода программы, на самом деле речь идет о защите данных, с которыми работает эта часть кода. То есть взаимное исключение обычно используется для защиты совместно используемых несколькими потоками или процессами данных.

Взаимные исключения представляют собой блокировку коллективного пользования. Это значит, что если совместно используемые данные представляют собой, например, связный список, то все потоки, работающие с этим списком, должны блокировать взаимное исключение. Ничто не может помешать потоку работать со списком, не заблокировав взаимное исключение. Взаимные исключения предполагают добровольное сотрудничество потоков.

7.3. Схема производитель-потребитель

Одна из классических задач на синхронизацию называется задачей производителя и потребителя. Она также известна как задача ограниченного буфера. Один или несколько производителей (потоков или процессов) создают данные, которые обрабатываются одним или несколькими потребителями. Эти данные передаются между производителями и потребителями с помощью одной из форм IPC.

С этой задачей мы регулярно сталкиваемся при использовании каналов Unix. Команда интерпретатора, использующая канал

является примером такой задачи. Программа gperf выступает как производитель (единственный), а wc — как потребитель (тоже единственный). Канал используется как форма IPC. Требуемая синхронизация между производителем и потребителем обеспечивается ядром, обрабатывающим команды write производителя и read покупателя. Если производитель опережает потребителя (канал переполняется), ядро приостанавливает производителя при вызове write, пока в канале не появится место. Если потребитель опережает производителя (канал опустошается), ядро приостанавливает потребителя при вызове read, пока в канале не появятся данные.

Такой тип синхронизации называется неявным; производитель и потребитель не знают о том, что синхронизация вообще осуществляется. Если бы мы использовали очередь сообщений Posix или System V в качестве средства IPC между производителем и потребителем, ядро снова взяло бы на себя обеспечение синхронизации.

При использовании разделяемой памяти как средства IPC производителя и потребителя, однако, требуется использование какого-либо вида явной синхронизации. Мы продемонстрируем это на использовании взаимного исключения. Схема рассматриваемого примера изображена на рис. 7.1.

В одном процессе у нас имеется несколько потоков-производителей и один поток-потребитель. Целочисленный массив buff содержит производимые и потребляемые данные (даны совместного пользования). Для простоты производители просто устанавливают значение buff[0] в 0, buff [1] в 1 и т.д. Потребитель перебирает элементы массива, проверяя правильность записей.

В этом первом примере мы концентрируем внимание на синхронизации между отдельными потоками-производителями. Поток-потребитель не будет запущен, пока все производители не завершат свою работу. В листинге 7.1 приведена функция main нашего примера.



Рис. 7.1. Производители и потребитель

4-12 Эти переменные совместно используются потоками. Мы объединяем их в структуру с именем shared вместе с взаимным исключением, чтобы подчеркнуть, что доступ к ним можно получить только вместе с ним. Переменная nput хранит индекс следующего элемента массива buff, подлежащего обработке, а nval содержит следующее значение, которое должно быть в него помещено (0, 1, 2 и т.д.). Мы выделяем память под эту структуру и инициализируем взаимное исключение, используемое для синхронизации потоков-производителей.

ПРИМЕЧАНИЕ

Мы всегда будем стараться размещать совместно используемые данные вместе со средствами синхронизации, к ним относящимися (взаимными исключениями, условными переменными, семафорами), в одной структуре, как мы сделали в этом примере. Это хороший стиль программирования. Однако во многих случаях совместно используемые данные являются динамическими, представляя собой, например, связный список. Мы, наверное, сможем поместить в структуру первый элемент списка вместе со средствами синхронизации (как в структуре mq_hdr в листинге 5.16), но оставшаяся часть списка в структуру не попадет. Следовательно, это решение не всегда является идеальным.

19-22 Первый аргумент командной строки указывает количество элементов, которые будут произведены производителями, а второй — количество запускаемых потоков-производителей.

23 Функция set_concurrency (наша собственная) указывает подсистеме потоков количество

одновременно выполняемых потоков. В Solaris 2.6 она просто вызывает `thr_setconcurrency`, причем ее запуск необходим, если мы хотим, чтобы у нескольких процессов-производителей была возможность начать выполняться. Если мы не сделаем этого вызова в системе Solaris, будет запущен только первый поток. В Digital Unix 4.0B наша функция `set_concurrency` не делает ничего, поскольку в этой системе по умолчанию все потоки процесса имеют равные права на вычислительные ресурсы.

ПРИМЕЧАНИЕ

Unix 98 требует наличия функции `pthread_setconcurrency`, выполняющей это же действие. Эта функция требуется для тех реализаций, которые мультиплексируют пользовательские потоки (создаваемые функцией `pthread_create`) на небольшое множество выполняемых потоков ядра. Такие реализации часто называются «многие-к-немногим» (many-to-few), «двухуровневые» (two-level) или «M-на-N» (M-to-N). В разделе 5.6 книги [3] отношения между пользовательскими потоками и потоками ядра рассматриваются более подробно.

24-28 Создаются потоки-производители, каждый из которых вызывает функцию `produce`. Идентификаторы потоков хранятся в массиве `tid_produce`. Аргументом каждого потока-производителя является указатель на элемент массива `count`. Счетчики инициализируются значением 0, и каждый поток увеличивает значение своего счетчика на 1 при помещении очередного элемента в буфер. Содержимое массива счетчиков затем выводится на экран, так что мы можем узнать, сколько элементов было помещено в буфер каждым из потоков.

29-36 Мы ожидаем завершения работы всех потоков-производителей, выводя содержимое счетчика для каждого потока, а затем запускаем единственный процесс-потребитель. Таким образом (на данный момент) мы исключаем необходимость синхронизации между потребителем и производителями. Мы ждем завершения работы потребителя, а затем завершаем работу процесса. В листинге 7.2 приведен текст функций `produce` и `consume`.

42-53 Критическая область кода производителя состоит из проверки на достижение конца массива (завершение работы)

и трех строк, помещающих очередное значение в массив:

Мы защищаем эту область с помощью взаимного исключения, не забыв разблокировать его после завершения работы. Обратите внимание, что увеличение элемента `count` (через указатель `arg`) не относится к критической области, поскольку у каждого потока счетчик свой (массив `count` в функции `main`). Поэтому мы не включаем эту строку в блокируемую взаимным исключением область. Один из принципов хорошего стиля программирования заключается в минимизации объема кода, защищаемого взаимным исключением.

59-62 Потребитель проверяет правильность значений всех элементов массива и выводит сообщение в случае обнаружения ошибки. Как уже говорилось, эта функция запускается в единственном экземпляре и только после того, как все потоки-производители завершат свою работу, так что надобность в синхронизации отсутствует.

При запуске только что описанной программы с пятью процессами-производителями, которые должны вместе создать один миллион элементов данных, мы получим следующий результат:

Как мы отмечали ранее, если убрать вызов `set_concurrency`, в системе Solaris 2.6 значение `count[0]` будет 1000000, а все остальные счетчики будут нулевыми.

Если убрать из этого примера блокировку с помощью взаимного исключения, он перестанет работать, как и предполагается. Потребитель обнаружит множество элементов `buff[i]`, значения которых будут отличны от `i`. Также мы можем убедиться, что удаление блокировки ничего не изменит, если будет выполняться только один поток.

7.4. Блокировка и ожидание

Продемонстрируем теперь, что взаимные исключения предназначены для блокирования, но не для ожидания. Изменим наш пример из предыдущего раздела таким образом, чтобы потребитель запускался сразу же после запуска всех производителей. Это даст возможность потребителю обрабатывать данные по мере их формирования производителями в отличие от программы в листинге 7.1, в которой потребитель не запускался до тех пор, пока все производители не завершили свою работу. Теперь нам придется синхронизовать потребителя с производителями, чтобы первый обрабатывал только данные, уже сформированные последними.

В листинге 7.3 приведен текст функции main. Начало кода (до объявления функции main) не претерпело никаких изменений по сравнению с листингом 7.1.

24 Мы увеличиваем уровень параллельного выполнения на единицу, чтобы учесть поток-потребитель, выполняемый параллельно с производителями.

25-29 Поток-потребитель создается сразу же после создания потоков-производителей.

Функция produce по сравнению с листингом 7.2 не изменяется. В листинге 7.4 приведен текст функции consume,зывающей новую функцию consume_wait.

71 Единственное изменение в функции consume заключается в добавлении вызова consume_wait перед обработкой следующего элемента массива.

57-64 Наша функция consume_wait должна ждать, пока производители не создадут i-й элемент. Для проверки этого условия производится блокировка взаимного исключения и значение i сравнивается с индексом производителя prut. Блокировка необходима, поскольку значение prut может быть изменено одним из производителей в момент его проверки.

Главная проблема — что делать, если нужный элемент еще не готов. Все, что нам остается и что мы делаем в листинге 7.4, — это повторять операции в цикле, устанавливая и снимая блокировку и проверяя значение индекса. Это называется опросом (spinning или polling) и является лишней тратой времени процессора.

Мы могли бы приостановить выполнение процесса на некоторое время, но мы не знаем, на какое. Что нам действительно нужно — это использовать какое-то другое средство синхронизации, позволяющее потоку или процессу приостанавливать работу, пока не произойдет какое-либо событие.

Взаимное исключение используется для блокирования, а условная переменная — для ожидания. Это два различных средства синхронизации, и оба они нужны. Условная переменная представляет собой переменную типа `pthread_cond_t`. Для работы с такими переменными предназначены две функции:

Слово `signal` в имени второй функции не имеет никакого отношения к сигналам Unix SIGxxx.

Мы определяем условие, уведомления о выполнении которого будем ожидать.

Взаимное исключение всегда связывается с условной переменной. При вызове `pthread_cond_wait` для ожидания выполнения какого-либо условия мы указываем адрес условной переменной и адрес связанного с ней взаимного исключения.

Мы проиллюстрируем использование условных переменных, переписав пример из предыдущего раздела. В листинге 7.5 объявляются глобальные переменные.

7-13 Две переменные `put` и `rival` ассоциируются с `mutex`, и мы объединяем их в структуру с именем `put`. Эта структура используется производителями.

14-20 Другая структура, `nready`, содержит счетчик, условную переменную и взаимное исключение. Мы инициализируем условную переменную с помощью `PTHREAD_COND_INITIALIZER`.

Функция `main` по сравнению с листингом 7.3 не изменяется.

Функции `produce` и `consume` претерпевают некоторые изменения. Их текст дан в листинге 7.6.

50-58 Для блокирования критической области в потоке-производителе теперь используется исключение `put.mutex`.

59-64 Мы увеличиваем счетчик `nready.nready`, в котором хранится количество элементов, готовых для обработки потребителем. Перед его увеличением мы проверяем, не было ли значение счетчика нулевым, и если да, то вызывается функция `pthread_cond_signal`, позволяющая возобновить выполнение всех потоков (в данном случае потребителя), ожидающих установки ненулевого значения этой переменной. Теперь мы видим, как взаимодействуют взаимное исключение и связанная с ним условная переменная. Счетчик используется совместно потребителем и производителями, поэтому доступ к нему осуществляется с блокировкой соответствующего взаимного исключения (`nready.mutex`). Условная переменная используется для ожидания и передачи сигнала.

72-76 Потребитель просто ждет, пока значение счетчика `nready.nready` не станет отличным от нуля. Поскольку этот счетчик используется совместно с производителями, его значение можно проверять только при блокировке соответствующего взаимного исключения. Если при проверке значение оказывается нулевым, мы вызываем `pthread_cond_wait` для приостановки процесса. При этом выполняются два атомарных действия:

1. Разблокируется `nready.mutex`.
2. Выполнение потока приостанавливается, пока какой-нибудь другой поток не вызовет `pthread_cond_signal` для этой условной переменной.

Перед возвращением управления потоку функция `pthread_cond_wait` блокирует `nready.mutex`. Таким образом, если после возвращения из функции мы обнаруживаем, что счетчик имеет ненулевое значение, мы уменьшаем этот счетчик (зная, что взаимное исключение заблокировано) и разблокируем взаимное исключение. Обратите внимание, что после возвращения из `pthread_cond_wait` мы всегда заново проверяем условие, поскольку может произойти ложное пробуждение при отсутствии выполнения условия. Различные реализации стремятся уменьшить количество ложных пробуждений, но они все равно происходят.

Код, передающий сигнал условной переменной, выглядит следующим образом:

В нашем примере переменная, для которой устанавливалось условие, представляла собой

целочисленный счетчик, а установка условия означала просто увеличение счетчика. Мы оптимизировали программу, посыпая сигнал только при изменении значения счетчика с 0 на 1.

Код, проверяющий условие и приостанавливающий процесс, если оно не выполняется, обычно выглядит следующим образом:

Исключение конфликтов блокировок

В приведенном выше фрагменте кода, как и в листинге 7.6, функция `pthread_cond_signal` вызывалась потоком, блокировавшим взаимное исключение, относящееся к условной переменной, для которой отправлялся сигнал. Мы можем представить себе, что в худшем варианте система немедленно передаст управление потоку, которому направляется сигнал, и он начнет выполняться и немедленно остановится, поскольку не сможет заблокировать взаимное исключение.

Альтернативный код, помогающий этого избежать, для листинга 7.6 будет иметь следующий вид:

Здесь мы отправляем сигнал условной переменной только после разблокирования взаимного исключения. Это разрешено стандартом Posix: поток,зывающий `pthread_cond_signal`, не обязательно должен в этот момент блокировать связанное с переменной взаимное исключение. Однако Posix говорит, что если требуется предсказуемое поведение при одновременном выполнении потоков, это взаимное исключение должно быть заблокировано процессом,зывающим `pthread_cond_signal`.

7.6. Условные переменные: время ожидания и широковещательная передача

В обычной ситуации `pthread_cond_signal` запускает выполнение одного потока, ожидающего сигнал по соответствующей условной переменной. В некоторых случаях поток знает, что требуется пробудить несколько других процессов. Тогда можно воспользоваться функцией `pthread_cond_broadcast` для пробуждения всех процессов, заблокированных в ожидании сигнала данной условной переменной.

ПРИМЕЧАНИЕ

Пример ситуации, в которой требуется возобновление выполнения нескольких процессов, появится в главе 8, когда мы будем обсуждать задачу с несколькими считывающими и записывающими процессами. Когда записывающий процесс снимает блокировку, он должен разбудить все ждущие считывающие процессы, поскольку одновременное считывание в данном случае разрешено.

Альтернативным (и более безопасным) методом является использование широковещательной передачи во всех тех случаях, когда требуется использование сигналов. Сигнал является оптимизацией для тех случаев, когда известно, что все ожидающие процессы правильно написаны и требуется разбудить только один из них, и какой именно будет разбужен, значения не имеет. Во всех других ситуациях следует использовать широковещательную передачу.

Функция `pthread_cond_timedwait` позволяет установить ограничение на время блокирования процесса. Аргумент *abstime* представляет собой структуру `timespec`:

Эта структура задает конкретный момент системного времени, в который происходит возврат из функции, даже если сигнал по условной переменной еще не будет получен. В этом случае возвращается ошибка с кодом `ETIMEDOUT`.

Эта величина представляет собой абсолютное значение времени, а не промежуток. Аргумент *abstime* задает таким образом количество секунд и наносекунд с 1 января 1970 UTC до того момента времени, в который должен произойти возврат из функции. Это отличает функцию от `select`, `pselect` и `poll` (глава 6 [24]), которые в качестве аргумента принимают некоторое количество долей секунды, спустя которое должен произойти возврат. (Функция `select` принимает количество микросекунд, `pselect` — наносекунд, а `poll` — миллисекунд.) Преимущество использования абсолютного времени заключается в том, что если функция возвратится до ожидаемого момента (например, при перехвате сигнала), ее можно будет вызвать еще раз, не изменяя содержимого структуры `timespec`.

В наших примерах в этой главе мы хранили взаимные исключения и условные переменные как глобальные данные всего процесса, поскольку они использовались для синхронизации потоков внутри него. Инициализировали мы их с помощью двух констант: PTHREAD_MUTEX_INITIALIZER и PTHREAD_COND_INITIALIZER. Инициализируемые таким образом исключения и условные переменные приобретали значения атрибутов по умолчанию, но мы можем инициализировать их и с другими значениями атрибутов.

Прежде всего инициализировать и удалять взаимное исключение и условную переменную можно с помощью функций

Рассмотрим, например, взаимное исключение. Аргумент mptr должен указывать на переменную типа `pthread_mutex_t`, для которой должна быть уже выделена память, и тогда функция `pthread_mutex_init` инициализирует это взаимное исключение. Значение типа `pthread_mutexattr_t`, на которое указывает второй аргумент функции `pthread_mutex_init(attr)`, задает атрибуты этого исключения. Если этот аргумент представляет собой нулевой указатель, используются значения атрибутов по умолчанию.

Атрибуты взаимного исключения имеют тип `pthread_mutexattr_t`, а условной переменной — `pthread_condattr_t`, инициализируются и уничтожаются с помощью следующих функций:

После инициализации атрибутов взаимного исключения или условной переменной для включения или выключения отдельных атрибутов используются отдельные функции. Например, один из атрибутов позволяет использовать данное взаимное исключение или условную переменную нескольким процессам (а не потокам одного процесса). Этот атрибут мы будем использовать в последующих главах. Его значение можно узнать и изменить с помощью следующих функций:

Две функции `get` возвращают текущее значение атрибута через целое, на которое указывает `valptr`, а две функции `set` устанавливают значение атрибута равным значению `value`. Значение `value` может быть либо `PTHREAD_PROCESS_PRIVATE`, либо `PTHREAD_PROCESS_SHARED`. Последнее также называется атрибутом совместного использования процессами.

ПРИМЕЧАНИЕ

Эта возможность поддерживается только в том случае, если константа `_POSIX_THREAD_PROCESS_SHARED` определена в заголовочном файле `<unistd.h>`. Она является дополнительной согласно Posix.1 и обязательной по Unix 98 (табл. 1.3).

Нижеследующий фрагмент кода показывает, как нужно инициализировать взаимное исключение, чтобы его можно было совместно использовать нескольким процессам:

Мы объявляем переменную `mattr` типа `pthread_mutexattr_t`, инициализируем ее значениями атрибутов по умолчанию, а затем устанавливаем атрибут `PTHREAD_PROCESS_SHARED`, позволяющий совместно использовать взаимное исключение нескольким процессам. Затем `pthread_mutex_init` инициализирует само исключение с соответствующими атрибутами. Количество разделяемой памяти, которое следует выделить под взаимное исключение, равно `sizeof(pthread_mutex_t)`.

Практически такая же последовательность команд (с заменой `mutex` на `cond`) позволяет установить атрибут `PTHREAD_PROCESS_SHARED` для условной переменной, хранящейся в разделяемой несколькими процессами памяти.

Пример совместно используемых несколькими процессами взаимных исключений и условных переменных был приведен в листинге 5.18.

Завершение процесса, заблокировавшего ресурс

Когда взаимное исключение используется совместно несколькими процессами, всегда существует возможность, что процесс будет завершен (возможно, принудительно) во время работы с заблокированным им ресурсом. Не существует способа заставить систему автоматически снимать блокировку во время завершения процесса. Мы увидим, что это свойственно и блокировкам чтения-записи, и семафорам Posix. Единственный тип блокировок, автоматически снимаемых системой при завершении процесса, — блокировки записей fcntl (глава 9). При использовании семафоров System V можно специально указать ядру, следует ли автоматически снимать блокировки при завершении работы процесса (функция SEM_UNDO, о которой будет говориться в разделе 11.3).

Поток также может быть завершен в момент работы с заблокированным ресурсом, если его выполнение отменит (cancel) другой поток или он сам вызовет pthread_exit. Последнему варианту не следует уделять много внимания, поскольку поток должен сам знать, блокирует ли он взаимное исключение в данный момент или нет, и в зависимости от этого вызывать pthread_exit. На случай отмены другим потоком можно предусмотреть обработчик сигнала, вызываемый при отмене потока, что продемонстрировано в разделе 8.5. Если же для потока возникают фатальные условия, это обычно приводит к завершению работы всего процесса. Например, если поток делает некорректную операцию с указателем, что приводит к отправке сигнала SIGSEGV, это приводит к остановке всего процесса, если сигнал не перехватывается, и мы возвращаемся к предыдущей ситуации с гибелью процесса, заблокировавшего ресурс.

Даже если бы система автоматически разблокировала ресурсы после завершения процесса, это не всегда решало бы проблему. Блокировка защищала критическую область, в которой, возможно, изменились какие-то данные. Если процесс был завершен посреди этой области, что стало с данными? Велика вероятность того, что возникнут несоответствия, если, например, новый элемент будет не полностью добавлен в связный список. Если бы ядро просто разблокировало взаимное исключение при завершении процесса, следующий процесс, обратившийся к списку, обнаружил бы, что тот поврежден.

В некоторых случаях автоматическое снятие блокировки (или счетчика — для семафора) при завершении процесса не вызывает проблем. Например, сервер может использовать семафор System V (с функцией SEM_UNDO) для подсчета количества одновременно обслуживаемых клиентов. Каждый раз при порождении процесса вызовом fork он увеличивает значение семафора на единицу, уменьшая его при завершении работы дочернего процесса. Если дочерний процесс завершил работу досрочно, ядро само уменьшит значение семафора. Пример, в котором автоматическое снятие блокировки ядром (а не уменьшение счетчика, как в вышеописанной ситуации) также не вызывает проблем, приведен в разделе 9.7. Демон блокирует один из файлов данных при записи в него и не снимает эту блокировку до завершения работы. Если кто-то попробует запустить копию демона, она завершит работу досрочно, когда обнаружит наличие блокировки на запись. Это гарантирует работу единственного экземпляра демона. Если же демон досрочно завершил работу, ядро само снимет блокировку, что позволит запустить копию демона.

7.8. Резюме

Взаимные исключения (mutual exclusion — mutex) используются для защиты критических областей кода, запрещая его одновременное выполнение несколькими потоками. В некоторых случаях потоку, заблокировавшему взаимное исключение, требуется дождаться выполнения какого-либо условия для выполнения последующих действий. В этом случае используется ожидание сигнала по условной переменной. Условная переменная всегда связывается с каким-либо взаимным исключением. Функция `pthread_cond_wait`, приостанавливающая работу процесса, разблокирует взаимное исключение перед остановкой работы и заново блокирует его при возобновлении работы процесса спустя некоторое время. Сигнал по условной переменной передается каким-либо другим потоком, и этот поток может разбудить либо только один произвольный поток из множества ожидающих (`pthread_cond_signal`), либо все их одновременно (`pthread_cond_broadcast`).

Взаимные исключения и условные переменные могут быть статическими. В этом случае они инициализируются также статически. Они могут быть и динамическими, что требует динамической инициализации. Динамическая инициализация дает возможность указать атрибуты, в частности атрибут совместного использования несколькими процессами, что действительно, если взаимное исключение или условная переменная находится в разделяемой этими процессами памяти.

Упражнения

1. Удалите взаимное исключение из листинга 7.2 и убедитесь, что программа работает неправильно, если одновременно запущено более одного производителя.
2. Что произойдет с листингом 7.1, если убрать вызов `Pthread_join` для потока-потребителя?
3. Напишите программу,зывающую `pthread_mutexattr_init` и `pthread_condattr_init` в бесконечном цикле. Следите за используемой этим процессом памятью с помощью какой-нибудь программы, например `ps`. Что происходит? Теперь добавьте вызовы `pthread_mutexattr_destroy` и `pthread_condattr_destroy` и убедитесь, что утечки памяти нет.
4. В программе из листинга 7.6 производитель вызывает `pthread_cond_signal` только при изменении `nready`,`nready` с 0 на 1. Чтобы убедиться в эффективности этой оптимизации, вызывайте `pthread_cond_signal` каждый раз, когда `nready`,`nready` увеличивается на 1, и выведите его значение в главном потоке после завершения работы потребителя.

8.1. Введение

Взаимное исключение используется для предотвращения одновременного доступа нескольких потоков к критической области. Критическая область кода обычно содержит операции считывания или изменения данных, используемых потоками совместно. В некоторых случаях можно провести различие между считыванием данных и их изменением.

В этой главе описываются блокировки чтения-записи, причем существует различие между получением такой блокировки для считывания и для записи. Правила действуют следующие:

1. Любое количество потоков могут заблокировать ресурс для считывания, если ни один процесс не заблокировал его на запись.
2. Блокировка чтения-записи может быть установлена на запись, только если ни один поток не заблокировал ресурс для чтения или для записи.

Другими словами, произвольное количество потоков могут считывать данные, если ни один поток не изменяет их в данный момент. Данные могут быть изменены, только если никто другой их не считывает и не изменяет.

В некоторых приложениях данныечитываются чаще, чем изменяются, поэтому такие приложения выигрывают в быстродействии, если при их реализации будут использованы блокировки чтения-записи вместо взаимных исключений. Возможность совместного считывания данных произвольным количеством процессов позволит выполнять операции параллельно, и при этом данные все равно будут защищены от других потоков на время изменения их данным потоком.

Такой вид совместного доступа к ресурсу также носит название совместно-исключающей блокировки (*shared-exclusive*), поскольку тип используемой блокировки на чтение называется совместной блокировкой (*shared lock*), а тип используемой блокировки на запись называется исключающей блокировкой (*exclusive lock*). Существует также специальное название для данной задачи (несколько считающих процессов и один записывающий): задача читателей и писателей (*readers and writers problem*), и говорят также о блокировке читателей и писателя (*readers-writer lock*). В последнем случае слово «читатель» специально употреблено во множественном числе, а «писатель» — в единственном, чтобы подчеркнуть сущность задачи.

ПРИМЕЧАНИЕ

Обычным примером, иллюстрирующим необходимость использования блокировок чтения-записи, является банковский счет. Считывать баланс со счета могут несколько потоков одновременно, но если какой-либо поток захочет изменить данные, ему придется подождать, пока считающие потоки не закончат свои операции, и только после этого ему будет разрешено вносить изменения. До окончания процесса записи никакие другие процессы не должны получить доступа к счету.

Функции, описываемые в этой главе, определены стандартом Unix 98, поскольку блокировки чтения-записи не были частью стандарта Posix.1 1996 года. Эти функции были разработаны группой производителей Unix, известной под названием Aspen Group, в 1995 году вместе с другими расширениями, которые еще не были определены Posix.1. Рабочая группа Posix (1003.1j) в настоящее время разрабатывает набор расширений Pthreads, включающий блокировки чтения-записи, который, хочется верить, совпадет с описываемым в этой главе.

8.2. Получение и сброс блокировки чтения-записи

Блокировка чтения-записи имеет тип `pthread_rwlock_t`. Если переменная этого типа является статической, она может быть проинициализирована присваиванием значения константы `PTHREAD_RWLOCK_INITIALIZER`.

Функция `pthread_rwlock_rdlock` позволяет заблокировать ресурс для чтения, причем вызвавший процесс будет заблокирован, если блокировка чтения-записи уже установлена записывающим процессом. Функция `pthread_rwlock_wrlock` позволяет заблокировать ресурс для записи, причем вызвавший процесс будет заблокирован, если блокировка чтения-записи уже установлена каким-либо другим процессом (читывающим или записывающим). Функция `pthread_rwlock_unlock` снимает блокировку любого типа (чтения или записи):

```
#include <pthread.h>
```

Следующие две функции производят попытку заблокировать ресурс для чтения или записи, но если это невозможно, возвращают ошибку с кодом `EBUSY`, вместо того чтобы приостановить выполнение вызвавшего процесса:

8.3. Атрибуты блокировки чтения-записи

Мы уже отмечали, что статическая блокировка может быть проинициализирована присваиванием ей значения PTHREAD_RWLOCK_INITIALIZER. Эти переменные могут быть проинициализированы и динамически путем вызова функции pthread_rwlock_init.

Когда поток перестает нуждаться в блокировке, он может вызвать pthread_rwlock_destroy:

Если при инициализации блокировки чтения-записи *attr* представляет собой нулевой указатель, атрибутам присваиваются значения по умолчанию. Для присваивания атрибутам других значений следует воспользоваться двумя нижеследующими функциями:

После инициализации объекта типа pthread_rwlockattr_t для установки или сброса отдельных атрибутов используются специальные функции. Единственный определенный на настоящее время атрибут — PTHREAD_PROCESS_SHARED, который указывает на то, что блокировка используется несколькими процессами, а не отдельными потоками одного процесса. Две приведенные ниже функции используются для получения и установки значения этого атрибута:

Первая функция возвращает текущее значение в целом, на которое указывает аргумент *valptr*. Вторая функция устанавливает значение этого атрибута равным *value*, которое может быть либо PTHREAD_PROCESS_PRIVATE, либо PTHREAD_PROCESS_SHARED.

Для реализации блокировок чтения-записи достаточно использовать взаимные исключения и условные переменные. В этом разделе мы рассмотрим одну из возможных реализаций, в которой предпочтение отдается ожидающим записи потокам. Это не является обязательным; возможны альтернативы.

ПРИМЕЧАНИЕ

Этот и последующие разделы данной главы содержат усложненный материал, который можно при первом чтении пропустить.

Другие реализации блокировок чтения записи заслуживают отдельного изучения. В разделе 7.1.2 книги [3] представлена реализация, в которой приоритет имеют ожидающие записи потоки и предусмотрена обработка отмены выполнения потока (о которой мы вскоре будем говорить подробнее). В разделе B.18.2.3.1 стандарта IEEE 1996 [8] представлена другая реализация, в которой предпочтение имеют ожидающие записи потоки и в которой также предусмотрена обработка отмены. В главе 14 книги [12] также приводится возможная реализация, в которой приоритет имеют ожидающие записи процессы. Реализация, приведенная в этом разделе, взята из пакета ACE (<http://www.cs.wustl.edu/~schmidt/ACE.html>), автором которого является Дуг Шмидт (Doug Schmidt). Аббревиатура ACE означает Adaptive Communications Environment. Во всех четырех реализациях используются взаимные исключения и условные переменные.

Тип данных pthread_rwlock_t

В листинге 8.1 приведен текст заголовочного файла `pthread_rwlock.h`, в котором определен основной тип `pthread_rwlock_t` и прототипы функций, работающих с блокировками чтения и записи. Обычно все это находится в заголовочном файле `<pthread.h>`.

3-13 Наш тип `pthread_rwlock_t` содержит одно взаимное исключение, две условные переменные, один флаг и три счетчика. Мы увидим, для чего все это нужно, когда будем разбираться с работой функций нашей программы. При просмотре или изменении содержимого этой структуры мы должны устанавливать блокировку `rw_mutex`. После успешной инициализации структуры полю `rw_magic` присваивается значение `RW_MAGIC`. Значение этого поля проверяется всеми функциями — таким образом гарантируется, что вызвавший поток передал указатель на проинициализированную блокировку. Оно устанавливается в 0 после уничтожения блокировки.

Обратите внимание, что в счетчике `rw_refcount` всегда хранится текущий статус блокировки чтения-записи: -1 обозначает блокировку записи (и только одна такая блокировка может существовать в любой момент времени), 0 обозначает, что блокировка доступна и может быть установлена, а любое положительное значение соответствует количеству установленных блокировок на чтение.

14-17 Мы также определяем константу для статической инициализации нашей структуры.

Функция pthread_rwlock_init

Первая функция, `pthread_rwlock_init`, динамически инициализирует блокировку чтения-записи. Ее текст приведен в листинге 8.2.

7-8 Присваивание атрибутов с помощью этой функции не поддерживается, поэтому мы проверяем, чтобы указатель `attr` был нулевым.

9-19 Мы инициализируем взаимное исключение и две условные переменные, которые содержатся в нашей структуре. Все три счетчика устанавливаются в 0, а полю `rw_magic` присваивается значение, указывающее на то, что структура была проинициализирована.

20-25 Если при инициализации взаимного исключения или условной переменной возникает ошибка, мы аккуратно уничтожаем проинициализированные объекты и возвращаем код ошибки.

Функция pthread_rwlock_destroy

В листинге 8.3 приведена функция pthread_rwlock_destroy, уничтожающая блокировку чтения записи после окончания работы с ней.

8-13 Прежде всего проверяется, не используется ли блокировка в данный момент, а затем вызываются соответствующие функции для уничтожения взаимного исключения и двух условных переменных.

Функция pthread_rwlock_rdlock

Текст функции pthread_rwlock_rdlock приведен в листинге 8.4.

9-10 При работе со структурой pthread_rwlock_t всегда устанавливается блокировка на rw_mutex, являющееся ее полем.

11-18 Нельзя получить блокировку на чтение, если rw_refcount имеет отрицательное значение (блокировка установлена на запись) или имеются потоки, ожидающие возможности получения блокировки на запись (rw_nwaitwriters больше 0). Если одно из этих условий верно, мы увеличиваем значение rw_nwaitreaders и вызываем pthread_cond_wait для условной переменной rw_condreaders. Вскоре мы увидим, что при разблокировании ресурса прежде всего проверяется наличие процессов, ожидающих возможности установить блокировку на запись, и если таковых не существует, проверяется наличие ожидающих возможности считывания. Если они имеются, для условной переменной rw_condreaders передается широковещательный сигнал.

19-20 При получении блокировки на чтение мы увеличиваем значение rw_refcount. Блокировка взаимного исключения после этого снимается.

ПРИМЕЧАНИЕ

В этой функции есть проблема: если вызвавший поток будет заблокирован в функции pthread_cond_wait и после этого его выполнение будет отменено, он завершит свою работу, не разблокировав взаимное исключение, и значение rw_nwaitreaders окажется неверным. Та же проблема есть и в функции pthread_rwlock_wrlock в листинге 8.6. Эти проблемы будут исправлены в разделе 8.5.

Функция pthread_rwlock_tryrdlock

В листинге 8.5 показана наша реализация функции pthread_rwlock_tryrdlock, которая не вызывает приостановления вызвавшего ее потока.

11-14 Если блокировка в данный момент установлена на запись или есть процессы, ожидающие возможности установить ее на запись, возвращается ошибка с кодом EBUSY. В противном случае мы устанавливаем блокировку, увеличивая значение счетчика rw_refcount.

Функция pthread_rwlock_wrlock

Текст функции pthread_rwlock_wrlock приведен в листинге 8.6.

11-17 Если ресурс заблокирован на считывание или запись (значение rw_refcount отлично от 0), мы приостанавливаем выполнение потока. Для этого мы увеличиваем rw_nwaitwriters и вызываем pthread_cond_wait с условной переменной rw_condwriters. Для этой переменной посыпается сигнал при снятии блокировки чтения-записи, если имеются ожидающие разрешения на запись процессы.

18-19 После получения блокировки на запись мы устанавливаем значение rw_refcount в -1.

Функция pthread_rwlock_trywrlock

Неблокируемая функция pthread_rwlock_trywrlock показана в листинге 8.7.

11-14 Если значение счетчика rw_refcount отлично от нуля, блокировка в данный момент уже установлена считывающим или записывающим процессом (это безразлично) и мы возвращаем ошибку с кодом EBUSY. В противном случае мы устанавливаем блокировку на запись, присвоив переменной rw_refcount значение -1.

Функция pthread_rwlock_unlock

Последняя функция, pthread_rwlock_unlock, приведена в листинге 8.8.

11-16 Если rw_refcount больше 0,читывающий поток снимает блокировку на чтение. Если rw_refcount равно -1, записывающий поток снимает блокировку на запись.

17-22 Если имеются ожидающие разрешения на запись потоки, по условной переменной rw_condwriters передается сигнал (если блокировка свободна, то есть значение счетчика rw_refcount равно 0). Мы знаем, что только один поток может осуществлять запись, поэтому используем функцию pthread_cond_signal. Если нет потоков, ожидающих возможности записи, но есть потоки, ожидающие возможности чтения, мы вызываем pthread_cond_broadcast для переменной rw_condreaders, поскольку возможно одновременное считывание несколькими потоками. Обратите внимание, что мы перестаем устанавливать блокировку для считающих потоков, если появляются потоки, ожидающие возможности записи. В противном случае постоянно появляющиеся потоки с запросами на чтение могли бы заставить поток, ожидающий возможности записи, ждать целую вечность. По этой причине мы используем два отдельных оператора if и не можем написать просто:

Мы могли бы исключить и проверку rw->rw_refcount, но это может привести к вызовам pthread_cond_signal даже при наличии блокировок на чтение, что приведет к потере эффективности.

Обсуждая листинг 8.4, мы обратили внимание на наличие проблемы, возникающей при отмене выполнения потока, заблокированного вызовом `pthread_cond_wait`. Выполнение потока может быть отменено в том случае, если какой-нибудь другой поток вызовет функцию `pthread_cancel`, единственным аргументом которой является идентификатор потока, выполнение которого должно быть отменено:

Отмена выполнения может быть использована в том случае, если несколько потоков начинают работу над какой-то задачей (например, поиск записи в базе данных) и один из них завершает работу раньше всех остальных. Тогда он может отменить их выполнение. Другим примером является обнаружение ошибки одним из одновременно выполняющих задачу потоков, который затем может отменить выполнение и всех остальных.

Для обработки отмены выполнения поток может установить (`push`) или снять (`pop`) обработчик-очиститель (`cleanup handler`):

Эти обработчики представляют собой обычные функции, которые вызываются:

- в случае отмены выполнения потока (другим потоком, вызвавшим `pthread_cancel`);
 - в случае добровольного завершения работы (вызовом `pthread_exit` или выходом из начальной функции потока).

Обработчики-очистители выполняют всю необходимую работу по восстановлению значений переменных, такую как разблокирование взаимных исключений и семафоров, которые могли быть заблокированы данным потоком.

Аргумент *function* представляет собой адрес вызываемой функции, а *arg* — ее единственный аргумент. Функция `pthread_cleanup_pop` всегда удаляет обработчик из верхушки стека и вызывает эту функцию, если значение *execute* отлично от 0.

ПРИМЕЧАНИЕ

Мы снова встретимся с проблемой отмены выполнения потоков в связи с листингом 15.26, где может произойти отмена выполнения сервера с дверьми при завершении работы клиента в процессе обработки вызванной им процедуры.

Пример

Легче всего продемонстрировать проблему нашей реализации из предыдущего раздела с помощью примера. На рис. 8.1 изображена временная диаграмма выполнения нашей программы, а текст самой программы приведен в листинге 8.9.



Рис. 8.1. Временная диаграмма выполнения программы из листинга 8.9

10-13 Создаются два потока, первый из которых выполняет функцию `thread1`, а второй — `thread2`. После создания первого делается пауза длительностью в одну секунду, чтобы он успел заблокировать ресурс на чтение.

14-23 Мы ожидаем завершения работы второго потока и проверяем, что его статус имеет значение `PTHREAD_CANCEL`. Затем мы ждем завершения работы первого потока и проверяем, что его статус представляет собой нулевой указатель. Затем мы выводим значение трех счетчиков в структуре `pthread_rwlock_t` и уничтожаем блокировку.

26-36 Поток получает блокировку на чтение и ждет 3 секунды. Эта пауза дает возможность другому потоку вызвать `pthread_rwlock_wrlock` и заблокироваться при вызове `pthread_cond_wait`, поскольку блокировка на запись не может быть установлена из-за наличия блокировки на чтение. Затем первый поток вызывает `pthread_cancel` для отмены выполнения второго потока, ждет 3 секунды, освобождает блокировку на чтение и завершает работу.

37-46 Второй поток делает попытку получить блокировку на запись (которую он получить не может, поскольку первый поток получил блокировку на чтение). Оставшаяся часть функции никогда не будет выполнена.

При запуске этой программы с использованием функций из предыдущего раздела мы получим следующий результат:

и мы никогда не вернемся к приглашению интерпретатора. Программа зависнет. Произошло вот что:

1. Второй поток вызвал `pthread_rwlock_wrlock` (листинг 8.6), которая была заблокирована в вызове `pthread_cond_wait`.
2. Первый поток вернулся из вызова `sleep(3)` и вызвал `pthread_cancel`.
3. Второй поток был отменен и завершил работу. При отмене потока, заблокированного в ожидании сигнала по условной переменной, взаимное исключение блокируется до вызова первого обработчика-очистителя. (Мы не устанавливали обработчик, но взаимное исключение все равно блокируется до завершения потока.) Следовательно, при отмене выполнения второго потока взаимное исключение осталось заблокированным и значение `rw_nwaitwriters` в листинге 8.6 было увеличено.
4. Первый поток вызывает `pthread_rwlock_unlock` и блокируется навсегда при вызове `pthread_mutex_lock` (листинг 8.8), потому что взаимное исключение все еще заблокировано отмененным потоком.

Если мы уберем вызов `pthread_rwlock_unlock` в функции `thread1`, функция `main` выведет вот что:

Первый счетчик имеет значение 1, поскольку мы удалили вызов `pthread_rwlock_unlock`, а последний счетчик имеет значение 1, поскольку он был увеличен вторым потоком до того, как тот был отменен.

Исправить эту проблему просто. Сначала добавим две строки к функции `pthread_rwlock_rdlock` в листинге 8.4. Строки отмечены знаком +:

Первая новая строка устанавливает обработчик-очиститель (функцию `rwlock_cancelrdwait`), а его

единственным аргументом является указатель `rw`. После возвращения из `pthread_cond_wait` вторая новая строка удаляет обработчик. Аргумент функции `pthread_cleanup_pop` означает, что функцию-обработчик при этом вызывать не следует. Если этот аргумент имеет ненулевое значение, обработчик будет сначала вызван, а затем удален.

Если поток будет отменен при вызове `pthread_cond_wait`, возврата из нее не произойдет. Вместо этого будут запущены обработчики (после блокирования соответствующего взаимного исключения, как мы отметили в пункте 3 чуть выше).

В листинге 8.10 приведен текст функции `rwlock_cancelrdwait`, являющейся обработчиком-очистителем для `pthread_rwlock_rdlock`.

8-9 Счетчик `rw_nwaitreaders` уменьшается, а затем разблокируется взаимное исключение. Это состояние, которое должно быть восстановлено при отмене потока.

Аналогично мы исправим текст функции `pthread_rwlock_wrlock` из листинга 8.6. Сначала добавим две новые строки рядом с вызовом `pthread_cond_wait`:

В листинге 8.11 приведен текст функции `rwlock_cancelrwait`, являющейся обработчиком-очистителем для запроса блокировки на запись.

8-9 Счетчик `rw_nwaitwriters` уменьшается, и взаимное исключение разблокируется. При запуске нашей тестовой программы из листинга 8.9 с этими новыми функциями мы получим правильные результаты:

Теперь три счетчика имеют правильные значения, первый поток возвращается из вызова `pthread_rwlock_unlock`, а функция `pthread_rwlock_destroy` не возвращает ошибку `EBUSY`.

ПРИМЕЧАНИЕ

Этот раздел представляет собой обзор вопросов, связанных с отменой выполнения потоков. Для более детального изучения этих проблем можно обратиться, например, к разделу 5.3 книги [3].

8.6. Резюме

Блокировки чтения-записи позволяют лучше распараллелить работу с данными, чем обычные взаимные исключения, если защищаемые данные чаще считаются, чем изменяются. Функции для работы с этими блокировками определены стандартом Unix 98, их мы и описываем в этой главе. Аналогичные или подобные им функции должны появиться в новой версии стандарта Posix. По виду функции аналогичны функциям для работы со взаимными исключениями (глава 7).

Блокировки чтения-записи легко реализовать с использованием взаимных исключений и условных переменных. Мы приводим пример возможной реализации. В нашей версии приоритет имеют записывающие потоки, но в некоторых других версиях приоритет может быть отдан ичитывающим потокам.

Потоки могут быть отменены в то время, когда они находятся в заблокированном состоянии, в частности при вызове `pthread_cond_wait`, и на примере нашей реализации мы убедились, что при этом могут возникнуть проблемы. Решить эту проблему можно путем использования обработчиков-очистителей.

Упражнения

1. Измените реализацию в разделе 8.4 таким образом, чтобы приоритет имели считывающие, а не записывающие потоки.
2. Сравните скорость работы нашей реализации из раздела 8.4 с предоставленной производителем.

9.1. Введение

Блокировки чтения-записи, описанные в предыдущей главе, представляют собой хранящиеся в памяти переменные типа `pthread_rwlock_t`. Эти переменные могут использоваться потоками одного процесса (этот режим работы установлен по умолчанию) либо несколькими процессами при условии, что переменные располагаются в разделяемой этими процессами памяти и при их инициализации был установлен атрибут `PTHREAD_PROCESS_SHARED`,

В этой главе описан усовершенствованный тип блокировки чтения-записи, который может использоваться родственными и неродственными процессами при совместном доступе к файлу. Обращение к блокирующему файлу осуществляется через его дескриптор, а функция для работы с блокировкой называется `fctl`. Такой тип блокировки обычно хранится в ядре, причем информация о владельце блокировки хранится в виде его идентификатора процесса. Таким образом, блокировки записей `fctl` могут использоваться только несколькими процессами, но не отдельными потоками одного процесса.

В этой главе мы в первый раз встретимся с нашим примером на увеличение последовательного номера. Рассмотрим следующую ситуацию, с которой столкнулись, например, разработчики спулера печати для Unix (команда `lpr` в BSD и `lp` в System V). Процесс, помещающий задания в очередь печати для последующей их обработки другим процессом, должен присваивать каждому из них уникальный последовательный номер. Идентификатор процесса, уникальный во время его выполнения, не может использоваться как последовательный номер, поскольку задание может просуществовать достаточно долго для того, чтобы этот идентификатор был повторно использован другим процессом. Процесс может также отправить на печать несколько заданий, каждому из которых нужно будет присвоить уникальный номер. Метод, используемый спулерами печати, заключается в том, чтобы хранить очередной порядковый номер задания для каждого принтера в отдельном файле. Этот файл содержит всего одну строку с порядковым номером в формате ASCII. Каждый процесс, которому нужно воспользоваться этим номером, должен выполнить следующие три действия:

1. Считать порядковый номер из файла.
2. Использовать этот номер.
3. Увеличить его на единицу и записать обратно в файл.

Проблема в том, что пока один процесс выполняет эти три действия, другой процесс может параллельно делать то же самое. В итоге возникнет полный беспорядок с номерами, как мы увидим в следующих примерах.

ПРИМЕЧАНИЕ

Описанная выше проблема называется проблемой взаимных исключений. Она может быть решена с использованием взаимных исключений из главы 7 или блокировок чтения-записи из главы 8. Различие состоит в том, что здесь мы предполагаем неродственность процессов, что усложняет использование предложенных выше методов. Мы могли бы использовать разделяемую память (подробно об этом говорится в четвертой части книги), поместив в нее переменную синхронизации одного из этих типов, но для неродственных процессов проще воспользоваться блокировкой `fctl`. Другим фактором в данном случае стало то, что проблема со спулерами печати возникла задолго до появления взаимных исключений, условных переменных и блокировок чтения-записи. Блокировка записей была добавлена в Unix в начале 80-х, до того как появились концепции разделяемой памяти и программных потоков.

Таким образом, процессу нужно заблокировать файл, чтобы никакой другой процесс не мог получить к нему доступ, пока первый выполняет свои три действия. В листинге 9.2 приведен текст простой программы, выполняющей соответствующие действия. Функции `my_lock` и `my_unlock` обеспечивают блокирование и разблокирование файла в соответствующие моменты. Мы приведем несколько возможных вариантов реализации этих функций.

20 Каждый раз при прохождении цикла мы выводим имя программы (`argv[0]`) перед порядковым номером, поскольку эта функция `main` будет использоваться с различными версиями функций блокировки и нам бы хотелось видеть, какая версия программы выводит данную последовательность порядковых номеров.

ПРИМЕЧАНИЕ

Вывод идентификатора процесса требует преобразования переменной типа `pid_t` к типу `long` и последующего использования строки формата `%ld`. Проблема тут в том, что идентификатор процесса принадлежит к одному из целых типов, но мы не знаем, к какому именно, поэтому предполагается наиболее вместительный — `long`. Если бы мы предположили, что идентификатор

имеет тип `int` и использовали бы строку `%d`, а `pid_t` на самом деле являлся бы типом `long`, код мог бы работать неправильно.

Посмотрим, что будет, если не использовать блокировку. В листинге 9.1 приведены версии функций `my_lock` и `my_unlock`, которые вообще ничего не делают.

Если начальное значение порядкового номера в файле было 1 и был запущен только один экземпляр программы, мы увидим следующий результат:

ПРИМЕЧАНИЕ

Обратите внимание, что функция `main` хранится в файле `lockmain.c`, но мы компилируем и компонуем эту программу с функциями, не осуществляющими никакой блокировки (листинг 9.1), поэтому мы называем ее `locknone`. Ниже будут использоваться другие версии функций `my_lock` и `my_unlock`, и исполняемый файл будет называться по-другому в соответствии с используемым методом блокировки.

Установим значение последовательного номера в файле обратно в единицу и запустим программу в двух экземплярах в фоновом режиме. Результат будет такой:

Первое, на что мы обращаем внимание, — подсказка интерпретатора, появившаяся до начала текста, выводимого программой. Это нормально и всегда имеет место при запуске программ в фоновом режиме.

Первые двадцать строк вывода не содержат ошибок. Они были сформированы первым экземпляром программы (с идентификатором 15 498). Проблема возникает в первой строке, выведенной вторым экземпляром (идентификатор 15499): он напечатал порядковый номер 1. Получилось это, скорее всего, так: второй процесс был запущен ядром, считал из файла порядковый номер (1), а затем управление было передано первому процессу, который работал до завершения. Затем второй процесс снова получил управление и продолжил выполняться с тем значением порядкового номера, которое было им уже считано (1). Это не то, что нам нужно. Каждый процесс считывает значение, увеличивает его и записывает обратно 20 раз (на экран выведено ровно 40 строк), поэтому конечное значение номера должно быть 40.

Нам нужно каким-то образом предотвратить изменение файла с порядковым номером на протяжении выполнения трех действий одним из процессов. Эти действия должны выполняться как атомарная операция по отношению к другим процессам. Код между вызовами `my_lock` и `my_unlock` представляет собой критическую область (глава 7).

При запуске двух экземпляров программы в фоновом режиме результат на самом деле непредсказуем. Нет никакой гарантии, что при каждом запуске мы будем получать один и тот же результат. Это нормально, если три действия будут выполняться как одна атомарная операция; в этом случае конечное значение порядкового номера все равно будет 40. Однако при неатомарном выполнении конечное значение часто будет отличным от 40, и это нас не устраивает. Например, нам безразлично, будет ли порядковый номер увеличен от 1 до 20 первым процессом и от 21 до 40 вторым или же процессы будут по очереди увеличивать его значение на единицу.

Неопределенность не делает результат неправильным, а вот атомарность выполнения операций — делает. Однако неопределенность выполнения усложняет отладку программ.

Ядро Unix никак не интерпретирует содержимое файла, оставляя всю обработку записей приложениям, работающим с этим файлом. Тем не менее для описания предоставляемых возможностей используется термин «блокировка записей». В действительности приложение указывает диапазон байтов файла для блокирования или разблокирования. Сколько логических записей помещается в этот диапазон — значения не имеет.

Стандарт Posix определяет один специальный диапазон с началом в 0 (начало файла) и длиной 0 байт, который устанавливает блокировку для всего файла целиком. Мы будем говорить о блокировке записей, подразумевая блокировку файла как частный случай.

Термин «степень детализации» (granularity) используется для описания минимального размера блокируемого объекта. Для стандарта Posix эта величина составляет 1 байт. Обычно степень детализации связана с максимальным количеством одновременных обращений к файлу. Пусть, например, с некоторым файлом одновременно работают пять процессов, из которых три считывают данные из файла и два записывают в него. Предположим также, что каждый процесс работает со своим набором записей и каждый запрос требует примерно одинакового времени для обработки (1 секунда). Если блокировка осуществляется на уровне файла (самый низкий уровень детализации), три считающих процесса смогут работать со своими записями одновременно, а двум записывающим придется ждать окончания их работы. Затем запись будет произведена сначала одним из оставшихся процессов, а потом другим. Полное затраченное время будет порядка 3 секунд (это, разумеется, очень грубая оценка). Если же уровень детализации соответствует размеру записи (наилучший уровень детализации), все пять процессов смогут работать одновременно, поскольку они обрабатывают разные записи. При этом на выполнение будет затрачена только одна секунда.

ПРИМЕЧАНИЕ

Потомки BSD поддерживают лишь блокировку файла целиком с помощью функции flock. Возможность заблокировать диапазон байтов не предусматривается.

История

За долгие годы было разработано множество методов блокировки файлов и записей. Древние программы вроде UUCP и демонов печати играли на реализации файловой системы (три из них описаны в разделе 9.8). Они работали достаточно медленно и не подходили для баз данных, которые стали появляться в начале 80-х.

Первый раз возможность блокировать файлы и записи появилась в Version 7, куда она была добавлена Джоном Бассом (John Bass) в 1980 году в виде нового системного вызова `locking`. Это блокирование было обязательным (*mandatory locking*); его унаследовали многие версии System III и Xenix. (Разница между обязательным и рекомендательным блокированием и между блокированием записей и файлов описана далее в этой главе.)

Версия 4.2BSD предоставила возможность блокирования файлов (а не записей) функцией `flock` в 1983. В 1984 году стандарт `/usr/group` (один из предшественников X/Open) определил функцию `lockf`, которая осуществляла только исключающую блокировку (на запись), но не совместную.

В 1984 году в System V Release 2 была добавлена возможность рекомендательной блокировки записей с помощью `fcntl`. Функция `lockf` в этой версии также имелась, но она осуществляла просто вызов `fcntl`. (Многие нынешние версии также реализуют `lockf` через вызов `fcntl`.) В 1986 году в версии System V Release 3 появилась обязательная блокировка записей с помощью `fcntl`. При этом использовался бит `set-group-ID` (установка идентификатора группы) — об этом методе рассказано в разделе 9.5.

В 1988 году стандарт Posix.1 включил в себя рекомендательную и обязательную блокировку файлов и записей с помощью функции `fcntl`, и это именно то, что является предметом обсуждения данной главы. Стандарт X/Open Portability Guide Issue 3 (XPG3, 1988) также указывает на необходимость осуществления блокировки записей через `fcntl`.

Согласно стандарту Posix, интерфейсом для блокировки записей является функция fcntl:

Для блокировки записей используются три различных значения аргумента *cmd*. Эти три значения требуют, чтобы третий аргумент, *arg*, являлся указателем на структуру flock:

Вот три возможные команды (значения аргумента *cmd*):

- F_SETLK — получение блокировки (*l_type* имеет значение либо F_RDLCK, либо F_WRLCK) или сброс блокировки (*l_type* имеет значение F_UNLCK), свойства которой определяются структурой flock, на которую указывает *arg*. Если процесс не может получить блокировку, происходит немедленный возврат с ошибкой EACCESS или EAGAIN.
- F_SETLKW — эта команда идентична предыдущей. Однако при невозможности блокирования ресурса процесс приостанавливается, до тех пор пока блокировка не сможет быть получена (W в конце команды означает «wait»).
- F_GETLK — проверка состояния блокировки, на которую указывает *arg*. Если в данный момент блокировка не установлена, поле *l_type* структуры flock, на которую указывает *arg*, будет иметь значение F_UNLCK. В противном случае в структуре flock, на которую указывает *arg*, возвращается информация об установленной блокировке, включая идентификатор процесса, заблокировавшего ресурс.

Обратите внимание, что последовательный вызов F_GETLK и F_SETLK не является атомарной операцией. Если мы вызвали F_GETLK и она вернула значение F_UNLCK в поле *l_type*, это не означает, что немедленный вызов F_SETLK будет успешным. Между этими двумя вызовами другой процесс мог уже заблокировать ресурс.

Причина, по которой была введена команда F_GETLK, — необходимость получения информации о блокировке в том случае, когда F_SETLK возвращает ошибку. Мы можем узнать, кто и каким образом заблокировал ресурс (на чтение или на запись). Но и в этом случае мы должны быть готовы к тому, что F_GETLK вернет результат F_UNLCK, поскольку между двумя вызовами другой процесс мог освободить ресурс.

Структура flock описывает тип блокировки (чтение или запись) и блокируемый диапазон. Как и в lseek, начальный сдвиг представляет собой сдвиг относительно начала файла, текущего положения или конца файла, и интерпретируется в зависимости от значения поля *l_whence* (SEEK_SET, SEEK_CUR, SEEK_END).

Поле *l_len* указывает длину блокируемого диапазона. Значение 0 соответствует блокированию от *l_start* до конца файла. Существуют, таким образом, два способа заблокировать файл целиком:

1. Указать *l_whence* = SEEK_SET, *l_start* = 0 и *l_len* = 0.
2. Перейти к началу файла с помощью lseek, затем указать *l_whence* = SEEK_CUR, *l_start* = 0 и *l_len* = 0.

Чаще всего используется первый метод, поскольку он предусматривает единственный вызов (fcntl — см. также упражнение 9.10).

Блокировка может быть установлена либо на чтение, либо на запись, и для конкретного байта файла может быть задан только один тип блокировки. Более того, на конкретный байт может быть установлено несколько блокировок на чтение, но только одна блокировка на запись. Это соответствует тому, что говорилось о блокировках чтения-записи в предыдущей главе. Естественно, при попытке установить блокировку на чтение для файла, открытого только на запись, будет возвращена ошибка.

Все блокировки, установленные конкретным процессом, снимаются при закрытии дескриптора файла этим процессом и при завершении его работы. Блокировки не наследуются дочерним процессом при вызове fork.

ПРИМЕЧАНИЕ

Снятие блокировок при завершении процесса обеспечивается только для блокировок записей fcntl и (в качестве дополнительной возможности) для семафоров System V. Для других средств синхронизации (взаимных исключений, условных переменных, блокировок чтения-записи и семафоров Posix) автоматическое снятие при завершении процесса не предусматривается. Об этом мы говорили в конце раздела 7.7.

Блокировка записей не должна использоваться со стандартной библиотекой ввода-вывода, поскольку функции из этой библиотеки осуществляют внутреннюю буферизацию. С заблокированными файлами следует использовать функции read и write, чтобы не возникало

неожиданных проблем.

Пример

Вернемся к нашему примеру из листинга 9.2 и перепишем функции `my_lock` и `my_unlock` из листинга 9.1 так, чтобы воспользоваться блокировкой записей Posix. Текст этих функций приведен в листинге 9.3.

Обратите внимание, что мы устанавливаем блокировку на запись, что гарантирует единственность изменяющего данные процесса (см. упражнение 9.4). При получении блокировки мы используем команду `F_SETLKW`, чтобы приостановить выполнение процесса при невозможности установки блокировки.

ПРИМЕЧАНИЕ

Зная определение структуры `flock`, приведенное выше, мы могли бы проинициализировать структуру `my_lock` как

но это неверно. Posix определяет только обязательные поля структуры, а реализации могут менять их порядок и добавлять к ним дополнительные.

Мы не приводим результат работы программы, но она, судя по всему, работает правильно. Выполнение этой программы не дает возможности утверждать, что в ней нет ошибок. Если результат оказывается неправильным, то можно сказать с уверенностью, что что-то не так. Но успешное выполнение программы еще ни о чем не говорит. Ядро могло выполнить сначала одну программу, затем другую, и если они не выполнялись параллельно, мы не имеем возможности увидеть ошибку. Увеличить шансы обнаружения ошибки можно, изменив функцию `main` таким образом, чтобы последовательный номер увеличивался 10000 раз, и запустив 20 экземпляров программы одновременно. Если начальное значение последовательного номера в файле было 1, мы можем ожидать, что после завершения работы всех этих процессов мы увидим в файле число 200001.

Пример: упрощение с помощью макросов

В листинге 9.3 установка и снятие блокировки занимали шесть строк кода. Мы должны выделить место под структуру, инициализировать ее и затем вызвать fcntl. Программы можно упростить, если определить следующие семь макросов, которые взяты из раздела 12.3 [21]:

Эти макросы используют наши функции lock_reg и lock_test, текст которых приведен в листингах 9.4 и 9.5. С ними нам уже не нужно заботиться об инициализации структур и вызове функций. Первые три аргумента специально сделаны совпадающими с первыми тремя аргументами функции lseek.

Мы также определяем две функции-обертки, Lock_reg и Lock_test, завершающие свое выполнение с возвратом ошибки fcntl, и семь макросов с именами, начинающимися с заглавной буквы, чтобы эти функции вызывать.

С помощью новых макросов мы можем записать функции my_lock и my_unlock из листинга 9.3 как

Блокировка записей по стандарту Posix называется рекомендательной. Ядро хранит информацию обо всех заблокированных различными процессами файлах, но оно не предотвращает запись в заблокированный на чтение процесс. Ядро также не предотвращает чтение из файла, заблокированного на запись. Процесс может игнорировать рекомендательную блокировку (*advisory lock*) и действовать по своему усмотрению (если у него имеются соответствующие разрешения на чтение и запись).

Рекомендательные блокировки отлично подходят для сотрудничающих процессов (*cooperating processes*). Примером сотрудничающих процессов являются сетевые демоны: все они находятся под контролем системного администратора. Пока в файл, содержащий порядковый номер, запрещена запись, никакой процесс не сможет его изменить.

Пример: несотрудничающие процессы

Мы можем проиллюстрировать рекомендательный характер блокировок, запустив два экземпляра нашей программы, один из которых (`lockfctl`) использует функции из листинга 9.3 и блокирует файл перед увеличением последовательного номера, а другой (`locknone`) использует функции из листинга 9.1 и не устанавливает никаких блокировок:

Программа `lockfctl` запускается первой, но в тот момент, когда она выполняет три действия для увеличения порядкового номера с 11 до 12 (в этот момент файл заблокирован), ядро переключается на второй процесс и запускает программу `locknone`. Этот процесс считывает значение 11 из файла с порядковым номером и использует его. Рекомендательная блокировка, установленная для этого файла программой `lockfctl`, никак не влияет на работу программы `locknone`.

Некоторые системы предоставляют возможность установки блокировки другого типа — обязательной (mandatory locking). В этом случае ядро проверяет все вызовы read и write, блокируя их при необходимости. Если для дескриптора установлен флаг O_NONBLOCK, вызов read или write, конфликтующий с установленной блокировкой, вернет ошибку EAGAIN. Если флаг O_NONBLOCK не установлен, выполнение процесса в такой ситуации будет отложено до тех пор, пока ресурс не освободится.

ПРИМЕЧАНИЕ

Стандарты Posix.1 и Unix 98 определяют только рекомендательную блокировку. Во многих реализациях, производных от System V, имеется возможность установки как рекомендательной, так и обязательной блокировки. Обязательная блокировка записей впервые появилась в System V Release 3.

Для установки обязательной блокировки какого-либо файла требуется выполнение двух условий:

- бит group-execute должен быть снят;
- бит set-group-ID должен быть установлен.

Обратите внимание, что установка бита set-user-ID без установки user-execute смысла не имеет; аналогично и с битами set-group-ID и group-execute. Таким образом, добавление возможности обязательной блокировки никак не повлияло на работу используемого программного обеспечения. Не потребовалось и добавлять новые системные вызовы.

В системах, поддерживающих обязательную блокировку записей, команда ls просматривает файлы на наличие указанной специальной комбинации битов и выводит буквы l или L, указывающие на то, что для данного файла включена обязательная блокировка. Аналогично команда chmod принимает аргумент l, позволяющий включить для указанного файла обязательную блокировку.

Пример

На первый взгляд, использование обязательной блокировки должно решить проблему с несогласующими процессами, поскольку все операции чтения и записи будут приостановлены до снятия блокировки. К сожалению, проблемы с одновременными обращениями к ресурсу являются гораздо более сложными, чем кажется, что мы можем легко продемонстрировать.

Чтобы использовать в нашем примере обязательную блокировку, изменим биты разрешений файла seqno. Кроме того, мы будем использовать новую версию функции main, которая принимает количество проходов цикла for в качестве аргумента командной строки (вместо использования константы 20) и не вызывает printf при каждом проходе цикла:

Теперь запустим две программы в качестве фоновых процессов: loopfcntl использует блокировку записей fcntl, а loopnone не использует блокировку вовсе.

Укажем в командной строке аргумента 10000 — количество последовательных увеличений порядкового номера.



Рис. 9.1. Временная диаграмма работы программ loopfcntl и loopnone

Каждый раз при выполнении этих программ результат будет между 14000 и 16000. Если бы блокировка работала так как надо, он всегда был бы равен 20001. Чтобы понять, где же возникает ошибка, нарисуем временную диаграмму выполнения процессов, изображенную на рис. 9.1.

Предположим, что loopfcntl запускается первой и выполняет первые восемь действий, изображенных на рисунке. Затем ядро передает управление другому процессу в тот момент, когда установлена блокировка на файл с порядковым номером. Запускается процесс loopnone, но он блокируется в первом вызове read, потому что на файл, который он пытается читать, установлена обязательная блокировка. Затем ядро передает управление первому процессу, который выполняет шаги 13-15. Пока все работает именно так, как мы предполагали, — ядро защищает файл от чтения несогласующим процессом, когда этот файл заблокирован.

Дальше ядро передает управление программе loopnone, которая выполняет шаги 17-23. Вызовы read и write разрешены, поскольку файл был разблокирован на шаге 15. Проблема возникает в тот момент, когда программа считывает значение 5 на шаге 23, а ядро в этот момент передает управление другому процессу. Он устанавливает блокировку и также считывает значение 5. Затем он дважды увеличивает это значение (получается 7), и управление передается loopnone на шаге 36. Однако эта программа записывает в файл значение 6. Так и возникает ошибка.

На этом примере мы видим, что обязательная блокировка предотвращает доступ к заблокированному файлу (шаг 11), но это не решает проблему. Проблема заключается в том, что левый процесс (на рисунке) может обновить содержимое файла (шаги 25-34) в тот момент, когда процесс справа также находится в состоянии обновления данных (шаги 23, 36 и 37). Если файл обновляется несколькими процессами, все они должны сотрудничать, используя некоторую форму блокировки. Один неподчиняющийся процесс может все разрушить.

В нашей реализации блокировок чтения-записи в разделе 8.4 приоритет предоставлялся ожидающим записи процессам. Теперь мы изучим детали возможного решения задачи читателей и писателей с помощью блокировки записей fcntl. Хочется узнать, как обрабатываются накапливающиеся запросы на блокировку, когда ресурс уже заблокирован. Стандарт Posix этого никак не оговаривает.

Пример: блокировка на чтение при наличии в очереди блокировки на запись

Первый вопрос, на который мы попытаемся найти ответ, звучит так: если ресурс заблокирован на чтение и какой-то процесс послал запрос на установление блокировки на запись, будет ли при этом разрешена установка еще одной блокировки на чтение? Некоторые решения задачи читателей и писателей не разрешают установки еще одной блокировки на чтение в случае, если есть ожидающий своей очереди пишущий процесс, поскольку если бы разрешалось непрерывное подключение считывающих процессов, запрос на запись, возможно, никогда бы не был удовлетворен.

Чтобы проверить, как эта ситуация разрешится при использовании блокировки записей fcntl, напишем тестовую программу, устанавливающую блокировку на чтение для всего файла и затем порождающую два процесса с помощью fork.

Первый из них пытается установить блокировку на запись (и блокируется, поскольку родительский процесс установил блокировку на чтение для всего файла), а второй процесс секунду спустя пытается получить блокировку на чтение. Временная диаграмма этих запросов изображена на рис. 9.2, а в листинге 9.6 приведен текст нашей программы.



Рис. 9.2. Определение возможности установки блокировки на чтение при наличии в очереди блокировки на запись

6-8 Родительский процесс открывает файл и устанавливает блокировку на чтение для всего файла целиком. Обратите внимание, что мы вызываем read_lock (которая возвращает ошибку в случае недоступности ресурса), а не readw_lock (которая ждет его освобождения), потому что мы ожидаем, что эта блокировка будет установлена немедленно. Мы также выводим значение текущего времени функцией gf_time [24, с. 404], когда получаем блокировку.

9-19 Порождается первый процесс, который ждет 1 секунду и блокируется в ожидании получения блокировки на запись для всего файла. Затем он устанавливает эту блокировку, ждет 2 секунды, снимает ее и завершает работу.

20-30 Порождается второй процесс, который ждет 3 секунды, давая возможность первому попытаться установить блокировку на запись, а затем пытается получить блокировку на чтение для всего файла. По моменту возвращения из функции readw_lock мы можем узнать, был ли ресурс предоставлен немедленно или второму процессу пришлось ждать первого. Блокировка снимается через четыре секунды.

31-35 Родительский процесс ждет пять секунд, снимает блокировку и завершает работу.

На рис. 9.2 приведена временная диаграмма выполнения программы в Solaris 2.6, Digital Unix 4.0B и BSD/OS 3.1. Как видно, блокировка чтения предоставляется второму дочернему процессу немедленно, несмотря на наличие в очереди запроса на блокировку записи. Существует вероятность, что запрос на запись так и не будет выполнен, если будут постоянно поступать новые запросы на чтение. Ниже приведен результат выполнения программы, в который были добавлены пустые строки для улучшения читаемости:

Пример: имеют ли приоритет запросы на запись перед запросами на чтение?

Следующий вопрос, на который мы попытаемся дать ответ, таков: есть ли приоритет у запросов на блокировку записи перед запросами на блокировку чтения, если все они находятся в очереди? Некоторые решения задачи читателей и писателей предусматривают это.

В листинге 9.7 приведен текст нашей тестовой программы, а на рис. 9.3 — временная диаграмма ее выполнения.

6-8 Родительский процесс создает файл и блокирует его целиком на запись.

9-19 Порождается первый процесс, который ждет одну секунду, а затем запрашивает блокировку на запись для всего файла. Мы знаем, что при этом процесс будет заблокирован, поскольку родительский процесс установил блокировку и снимет ее только через пять секунд, и мы хотим, чтобы этот запрос был помещен в очередь.

20-30 Порождается второй процесс, который ждет три секунды, а затем запрашивает блокировку на чтение на весь файл. Этот запрос будет также помещен в очередь.

И в Solaris 2.6, и в Digital Unix 4.0B мы видим, что блокировка на запись предоставляется первому процессу, как изображено на рис. 9.3. Но это еще не означает, что у запросов на запись есть приоритет перед запросами на чтение, поскольку, возможно, ядро предоставляет блокировку в порядке очереди вне зависимости от того, на чтение она или на запись. Чтобы проверить это, мы создаем еще одну тестовую программу, практически идентичную приведенной в листинге 9.7, но в ней блокировка на чтение запрашивается через одну секунду, а блокировка на запись — через три секунды. Эти две программы иллюстрируют, что Solaris и Digital Unix обрабатывают запросы в порядке очереди вне зависимости от типа запроса. Однако в BSD/OS 3.1 приоритет имеют запросы на чтение.



Рис. 9.3. Есть ли у писателей приоритет перед читателями

Вот вывод программы из листинга 9.7, на основании которого была составлена временная диаграмма на рис. 9.3:

9.7. Запуск единственного экземпляра демона

Часто блокировки записей используются для обеспечения работы какой-либо программы (например, демона) в единственном экземпляре. Фрагмент кода, приведенный в листинге 9.8, должен выполняться при запуске демона.

8-17 Демон создает однострочный файл, в который записывает свой идентификатор процесса. Этот файл открывается или создается, а затем делается попытка заблокировать его на запись целиком. Если блокировку установить не удается, мы понимаем, что один экземпляр демона уже запущен, поэтому выводится сообщение об ошибке и программа завершает работу.

ПРИМЕЧАНИЕ

Во многих версиях Unix демоны записывают свои идентификаторы в файл. Solaris 2.6 хранит подобные файлы в каталоге /etc, а Digital Unix 4.0B и BSD/OS — в каталоге /var/run.

18-21 Мы укорачиваем файл до 0 байт, а затем записываем в него строку с нашим идентификатором. Причина, по которой нужно укорачивать файл, заключается в том, что у предыдущего экземпляра демона идентификатор мог быть представлен более длинным числом, чем у данного, поэтому в результате в файле может образоваться смесь двух идентификаторов.

Вот результат работы программы из листинга 9.8:

Существуют и другие способы предотвращения запуска нескольких экземпляров демонов, например семафоры. Преимущество данного метода в том, что многим демонам и так приходится записывать в файл свои идентификаторы, а при досрочном завершении работы демона блокировка с файла снимается автоматически.

9.8. Блокирование файлов

Стандарт Posix.1 гарантирует, что если функция open вызывается с флагами O_CREAT (создать файл, если он еще не существует) и O_EXCL (исключающее открытие), функция возвращает ошибку, если файл уже существует. Более того, проверка существования файла и его создание (если он еще не существует) должны представлять собой атомарную по отношению к другим процессам операцию. Следовательно, мы можем использовать создаваемый таким методом файл как блокировку. Можно быть уверенными, что только один процесс сможет создать файл (то есть получить блокировку), а для снятия этой блокировки файл можно удалить командой unlink.

В листинге 9.9 приведен текст наших функций установки и снятия блокировки, использующих этот метод. При успешном выполнении функции open мы считаем, что блокировка установлена, и успешно возвращаемся из функции my_lock. Файл мы закрываем, потому что его дескриптор нам не нужен. О наличии блокировки свидетельствует само существование файла вне зависимости от того, открыт он или нет. Если функция open возвращает ошибку EEXIST, значит, файл существует и мы должны еще раз попытаться открыть его.

У этого метода есть три недостатка.

1. Если процесс, установивший блокировку, завершится досрочно, не сняв ее, файл не будет удален. Существуют способы борьбы с этой проблемой, например проверка времени доступа к файлу и удаление его спустя некоторый определенный промежуток времени, — но все они несовершенны. Другое решение заключается в записи в файл идентификатора процесса, чтобы другие процессы могли считать его и проверить, существует ли еще такой процесс. Этот метод также несовершенен, поскольку идентификатор может быть использован повторно.

В такой ситуации лучше пользоваться блокировкой fcntl, которая автоматически снимается по завершении процесса.

2. Если файл открыт каким-либо другим процессом, мы должны еще раз вызвать open, повторяя эти вызовы в бесконечном цикле. Это называется опросом и является напрасной тратой времени процессора. Альтернативным методом является вызов sleep на 1 секунду, а затем повторный вызов open (этап проблема обсуждалась в связи с листингом 7.4).

Эта проблема также исчезает при использовании блокировки fcntl, если использовать команду F_SETLKW. Ядро автоматически приостанавливает выполнение процесса до тех пор, пока ресурс не станет доступен.

3. Создание и удаление файла вызовом open и unlink приводит к обращению к файловой системе, что обычно занимает существенно больше времени, чем вызов fcntl (обращение производится дважды: один раз для получения блокировки, а второй — для снятия). При использовании fcntl программа выполнила 10000 повторов цикла с увеличением порядкового номера в 75 раз быстрее, чем программа, вызывавшая open и unlink.

Есть еще две особенности файловой системы Unix, которые использовались для реализации блокировок. Первая заключается в том, что функция link возвращает ошибку, если имя новой ссылки уже существует. Для получения блокировки создается уникальный временный файл, полное имя которого содержит в себе его идентификатор процесса (или комбинацию идентификаторов процесса и потока, если требуется осуществлять блокировку между отдельными потоками). Затем вызывается функция link для создания ссылки на этот файл с каким-либо определенным заранее именем. После успешного создания сам файл может быть удален вызовом unlink. После осуществления работы с блокировкой файл с известным именем удаляется командой unlink. Если link возвращает ошибку EEXIST, поток должен попытаться создать ссылку еще раз (аналогично листингу 9.9). Одно из требований к этому методу — необходимо, чтобы и временный файл, и ссылка находились в одной файловой системе, поскольку большинство версий Unix не допускают создания жестких ссылок (результат вызова link) в разных файловых системах.

Вторая особенность заключается в том, что функция open возвращает ошибку в случае существования файла, если указан флаг O_TRUNC и запрещен доступ на запись. Для получения блокировки мы вызываем open, указывая флаги O_CREAT | O_WRONLY | O_TRUNC и аргумент mode со значением 0 (то есть разрешения на доступ к файлу установлены в 0). Если вызов оказывается успешным, блокировка установлена и мы просто удаляем файл вызовом unlink после завершения работы. Если вызов open возвращает ошибку EACCESS, поток должен сделать еще одну попытку (аналогично листингу 9.9). Этот трюк не срабатывает, если поток обладает правами

привилегированного пользователя.

Урок, который можно извлечь из этих примеров, прост: нужно пользоваться блокировкой fcntl. Тем не менее вы можете столкнуться с программой, в которой используются старые методы блокировки, и особенно часто это будет встречаться в программах, написанных до широкого распространения реализации с блокировкой fcntl.

9.9. Блокирование в NFS

Аббревиатура NFS расшифровывается как Network File System (сетевая файловая система); эта система подробно обсуждается в главе 29 [22]. Блокировка записей fcntl представляет собой расширение NFS, поддерживаемое большинством ее реализаций. Обслуживается эта блокировка двумя дополнительными демонами: lockd и statd. При вызове fcntl для получения блокировки ядро обнаруживает, что файл находится в файловой системе NFS. Тогда локальный демон lockd посыпает демону lockd сервера запрос на получение блокировки. Демон statd хранит информацию о клиентах, установивших блокировку, и взаимодействует с lockd для обеспечения снятия блокировок в случае завершения процессов.

Установка блокировки записи в NFS должна занимать в среднем больше времени, чем для локального файла, поскольку для установки и снятия блокировки требуется передача информации по сети. Для проверки работы блокировки NFS нужно всего лишь изменить имя файла, определяемое константой SEQFILE в листинге 9.2. Если измерить время, требуемое для выполнения 10000 операций по увеличению порядкового номера новой версией программы, оно окажется примерно в 80 раз больше, чем для локального файла. Однако нужно понимать, что в этом случае происходит передача информации по сети и при операциях чтения и записи (для изменения порядкового номера).

ПРИМЕЧАНИЕ

Блокировка записей в NFS была связана с проблемами в течение многих лет, и большинство проблем были следствием плохой реализации. Несмотря на тот факт, что большинство производителей Unix все-таки доделали эту реализацию, использование блокировки fcntl через NFS все еще далеко от совершенства. Не будем делать безответственных утверждений: блокировка fcntl должна работать и в NFS, но будет ли — зависит от реализации демона и сервера.

9.10. Резюме

Блокирование записей с помощью `fcntl` предоставляет возможность установки рекомендательной или обязательной блокировки для файла, указанного с помощью открытого дескриптора. Эти блокировки предназначены для сотрудничества процессов, но не отдельных потоков одного процесса. Термин «запись» используется не вполне корректно, поскольку ядро не различает отдельные записи в файле. Лучше использовать термин «блокировка диапазона», поскольку при установке блокировки или ее снятии указывается именно диапазон байтов в файле. Практически во всех случаях применения этой блокировки она является рекомендательной и используется при совместной работе сотрудничающих процессов, поскольку даже обязательная блокировка не может исключить повреждения данных.

При использовании `fcntl` не гарантируется, что читающие или пишущие процессы имеют приоритет при ожидании (в отличие от того, что мы реализовали в главе 8 с блокировками чтения-записи). Если это важно для приложения, придется реализовать блокировки самостоятельно (как в разделе 8.4) с тем приоритетом, который требуется.

Упражнения

1. Создайте программу locknone из листингов 9.2 и 9.1 и выполните ее много раз. Убедитесь, что программа не работает и результат непредсказуем.
2. Измените листинг 9.2 так, чтобы стандартный поток вывода не буферизовался. Как это повлияет на работу программы?
3. Продолжайте изменять программу, вызывая putchar для каждого выводимого символа (вместо printf). Как изменится результат?
4. Измените блокировку в функции my_lock из листинга 9.3 так, чтобы устанавливалась блокировка на чтение, а не на запись. Что произойдет?
5. Измените вызов open в программе loopmain.c, указав также флаг O_NONBLOCK. Создайте программу loopfcntlnonb и запустите два экземпляра. Что произойдет?
6. Продолжите предыдущий пример, используя неблокируемую версию loopmain.c для создания программы loopnonenonb (используя файл locknone.c). Включите обязательную блокировку для файла seqno. Запустите один экземпляр этой программы и один экземпляр программы loopfcntlnonb из предыдущего примера одновременно. Что произойдет?
7. Создайте программу loopfcntl и запустите ее 10 раз в фоновом режиме из сценария интерпретатора команд. Каждому из 10 экземпляров следует указать аргумент 10000. Измерьте скорость работы сценария при использовании обязательной и рекомендательной блокировок. Как влияет обязательная блокировка на производительность?
8. Почему мы вызывали fork в листингах 9.6 и 9.7 для порождения процессов, вместо того чтобы воспользоваться pthread_create для создания потоков?
9. В листинге 9.9 мы вызываем ftruncate для установки размера файла в 0 байт. Почему бы нам просто не указать флаг O_TRUNC при открытии файла?
10. Какой из констант — SEEK_SET, SEEK_CUR или SEEK_END — следует пользоваться при указании блокируемого диапазона при написании многопоточного приложения и почему?

10.1. Введение

Семафор представляет собой простейшее средство синхронизации процессов и потоков. Мы рассматриваем три типа семафоров:

- именованные семафоры Posix, идентифицируемые именами, соответствующими стандарту Posix для IPC (см. раздел 2.2);
- размещаемые в разделяемой памяти семафоры Posix;
- семафоры System V (глава 11), обслуживаемые ядром.

Все три типа семафоров могут использоваться для синхронизации как отдельных процессов, так и потоков одного процесса. Мы начнем с рассмотрения проблем синхронизации между разными процессами с помощью бинарного семафора, то есть такого, который может принимать только значения 0 и 1. Пример подобной схемы приведен на рис. 10.1.



Рис. 10.1. Два процесса взаимодействуют с помощью бинарного семафора

На этом рисунке изображен бинарный семафор, хранящийся в ядре (семафор System V).

Семафоры Posix не обязательно обрабатываться ядром. Их особенностью является наличие имен, которые могут соответствовать именам реальных файлов в файловой системе. На рис. 10.2 изображена схема, лучше иллюстрирующая предмет обсуждения данной главы — именованный семафор Posix.



Рис. 10.2. Два процесса, использующие бинарный именованный семафор Posix

ПРИМЕЧАНИЕ

В отношении рис. 10.2 необходимо сделать одно уточнение: хотя именованные семафоры Posix обладают именами в файловой системе, они не обязательно должны храниться в файлах. Во встроенной системе реального времени значение семафора, скорее всего, будет размещаться в ядре, а имя файла будет использоваться исключительно для идентификации семафора. При реализации с помощью отображения файлов в память (пример такой реализации приведен в разделе 10.15) значение семафора будет действительно храниться в файле, который будет отображаться в адресное пространство всех процессов, использующих семафор.

На рис. 10.1 и 10.2 мы указали три операции, которые могут быть применены к семафорам:

1. Создание семафора. При этом вызвавший процесс должен указать начальное значение (часто 1, но может быть и 0).
2. Ожидание изменения значения семафора (*wait*). При этом производится проверка его значения и процесс блокируется, если значение оказывается меньшим либо равным 0, а при превышении 0 значение уменьшается на 1. Это может быть записано на псевдокоде как

Основным требованием является атомарность выполнения операций проверки значения в цикле *while* и последующего уменьшения значения семафора (то есть как одной операции) по отношению к другим потокам (это одна из причин, по которой семафоры System V были реализованы в середине 80-х как часть ядра. Поскольку операции с ними выполнялись с помощью системных вызовов, легко было гарантировать их атомарность по отношению к другим процессам).

У этой операции есть несколько общеупотребительных имен. Изначально она называлась *P*, от голландского *proeven* (проверка, попытка), — это название было введено Эдгером Дейкстрой. Используются также и термины *down* (поскольку значение семафора уменьшается) и *lock*, но мы будем следовать стандарту Posix и говорить об ожидании (*wait*).

3. Установка значения семафора (*post*). Значение семафора увеличивается одной командой, которая может быть записана на псевдокоде как

Если в системе имеются процессы, ожидающие изменения значения семафора до величины, превосходящей 0, один из них может быть пробужден. Как и операция ожидания, операция

установки значения семафора также должна быть атомарной по отношению к другим процессам, работающим с этим семафором.

Для этой операции также имеется несколько общеупотребительных терминов. Изначально она называлась *V*, от голландского *verhogen* (увеличивать). Называют ее *up* (значение семафора увеличивается), *unlock* и *signal*. Мы, следуя стандарту Posix, называем эту операцию *post*.

Очевидно, что реальный код для работы с семафором будет более сложным, чем приведенный выше. Все процессы, ожидающие изменения какого-либо семафора, должны помещаться в очередь, и один из них должен запускаться при выполнении требуемого условия. К счастью, это обеспечивается реализацией.

Обратите внимание, что приведенный псевдокод не ограничен в применении только бинарными семафорами. Код работает с семафором, инициализируемым любым неотрицательным значением. Такие семафоры называют также семафорами-счетчиками. Обычно они инициализируются некоторым значением *N*, которое указывает количество доступных ресурсов (например, буферов). В этой главе есть примеры использования как бинарных семафоров, так и семафоров-счетчиков.

ПРИМЕЧАНИЕ

Мы часто проводим различие между бинарными и многозначными семафорами, но делаем это исключительно в образовательных целях. В системной реализации семафоров никакой разницы нет.

Бинарный семафор может использоваться в качестве средства исключения (подобно взаимному исключению). В листинге 10.1 приведен пример для сравнения этих средств.

Мы инициализируем семафор значением 1. Вызвав *sem_wait*, мы ожидаем, когда значение семафора окажется больше 0, а затем уменьшаем его на 1. Вызов *sem_post* увеличивает значение с 0 до 1 и возобновляет выполнение всех потоков, заблокированных в вызове *sem_wait* для данного семафора.

Хотя семафоры и могут использоваться в качестве взаимных исключений, они обладают некоторыми особенностями: взаимное исключение должно быть разблокировано именно тем потоком, который его заблокировал, в то время как увеличение значения семафора может быть выполнено другим потоком. Можно привести пример использования этой особенности для решения упрощенной версии задачи потребителей и производителей из главы 7 с двумя бинарными семафорами. На рис. 10.3 приведена схема с одним производителем, помещающим объект в общий буфер, и одним потребителем, изымающим его оттуда. Для простоты предположим, что в буфер помещается ровно один объект.



Рис. 10.3. Задача производителя и потребителя с общим буфером

В листинге 10.2 приведен текст соответствующей программы на псевдокоде.

Семафор *put* ограничивает возможность помещения объекта в общий буфер, а семафор *get* управляет потребителем при считывании объекта из буфера. Работает эта программа в такой последовательности:

1. Производитель инициализирует буфер и два семафора.
2. Пусть после этого запускается потребитель. Он блокируется при вызове *sem_wait*, поскольку семафор *get* имеет значение 0.
3. После этого запускается производитель. При вызове *sem_wait* значение *put* уменьшается с 1 до 0, после чего производитель помещает объект в буфер. Вызовом *sem_post* значение семафора *get* увеличивается с 0 до 1. Поскольку имеется поток, заблокированный в ожидании изменения значения этого семафора, этот поток помечается как готовый к выполнению. Предположим, тем не менее, что производитель продолжает выполняться. В этом случае он блокируется при вызове *sem_wait* в начале цикла *for*, поскольку значение семафора *put* — 0. Производитель должен подождать, пока потребитель не извлечет данные из буфера.
4. Потребитель возвращается из *sem_wait*, уменьшая значение семафора *get* с 0 до 1. Затем он обрабатывает данные в буфере и вызывает *sem_post*, увеличивая значение *put* с 0 до 1. Заблокированный в ожидании изменения значения этого семафора поток-производитель помечается как готовый к выполнению. Предположим опять, что выполнение потребителя продолжается. Тогда он блокируется при вызове *sem_wait* в начале цикла *for*, поскольку семафор *get* имеет значение 0.

5. Производитель возвращается из `sem_wait`, помещает данные в буфер, и все повторяется.

Мы предполагали, что каждый раз при вызове `sem_post` продолжалось выполнение вызвавшего эту функцию потока, несмотря на то что ожидающий изменения значения семафора поток помечался как готовый к выполнению. Никаких изменений в работе программы не произойдет, если вместо вызвавшего `sem_post` потока будет выполняться другой, ожидавший изменения состояния семафора (исследуйте такую ситуацию и убедитесь в этом самостоятельно).

Перечислим три главных отличия семафоров и взаимных исключений в паре с условными переменными:

1. Взаимное исключение всегда должно разблокироваться тем потоком, который установил блокировку, тогда как увеличение значения семафора не обязательно осуществляется ожидающим его изменения потоком. Это мы только что продемонстрировали на примере.

2. Взаимное исключение может быть либо заблокировано, либо разблокировано (пара состояний, аналогично бинарному семафору).

3. Поскольку состояние семафора хранится в определенной переменной, изменение его значения оказывает влияние на процессы, которые вызовут функцию `wait` уже после этого изменения, тогда как при отправке сигнала по условной переменной в отсутствие ожидающих его потоков сигнал будет утерян. Взгляните на листинг 10.2 и представьте, что при первом проходе цикла производителем потребитель еще не вызвал `sem_wait`. Производитель сможет поместить объект в буфер, вызвать `sem_post` для семафора `get` (увеличивая его значение с 0 до 1), а затем он заблокируется в вызове `sem_wait` для семафора `put`. Через некоторое время потребитель дойдет до цикла `for` и вызовет `sem_wait` для переменной `get`, что уменьшит значение этого семафора с 1 до 0, а затем потребитель приступит к обработке содержимого буфера.

ПРИМЕЧАНИЕ

В Обосновании Posix.1 (Rationale) содержится следующий комментарий по поводу добавления семафоров помимо взаимных исключений и условных переменных: «Семафоры включены в стандарт в первую очередь с целью предоставить средства синхронизации выполнения процессов; эти процессы могут и не использовать общий сегмент памяти. Взаимные исключения и условные переменные описаны как средства синхронизации потоков, у которых всегда есть некоторое количество общей памяти. Оба метода широко используются уже много лет. Каждое из этих простейших средств имеет свой предпочтительный круг задач». В разделе 10.15 мы увидим, что для реализации семафоров-счетчиков с живучестью ядра требуется написать около 300 строк кода на C, использующего взаимные исключения и условные переменные. Несмотря на предпочтительность применения семафоров для синхронизации между процессами и взаимных исключений для синхронизации между потоками, и те и другие могут использоваться в обоих случаях. Следует пользоваться тем набором средств, который удобен в данном приложении.

Выше мы отмечали, что стандартом Posix описано два типа семафоров: именованные (`named`) и размещаемые в памяти (`memory-based` или `unnamed`). На рис. 10.4 сравниваются функции, используемые обоими типами семафоров.

Именованный семафор Posix был изображен на рис. 10.2. Неименованный, или размещаемый в памяти, семафор, используемый для синхронизации потоков одного процесса, изображен на рис. 10.5.



Рис. 10.4. Вызовы для семафоров Posix



Рис. 10.5. Семафор, размещенный в общей памяти двух потоков

На рис. 10.6 изображен размещенный в разделяемой памяти семафор (часть 4), используемый двумя процессами. Общий сегмент памяти принадлежит адресному пространству обоих процессов.



Рис. 10.6. Семафор, размещенный в разделяемой двумя процессами памяти

В этой главе сначала рассматриваются именованные семафоры Posix, а затем — размещаемые в памяти. Мы возвращаемся к задаче производителей и потребителей из раздела 7.3 и расширяем ее, позволяя некоторым производителям работать с одним потребителем, а в конце концов переходим к некоторым производителям и некоторым потребителям. Затем мы покажем, что часто используемый при реализации ввода-вывода метод множественных буферов является частным случаем задачи производителей и потребителей.

Мы рассмотрим три реализации именованных семафоров Posix: с использованием каналов FIFO, отображаемых в память файлов и семафоров System V.

10.2. Функции `sem_open`, `sem_close` и `sem_unlink`

Функция `sem_open` создает новый именованный семафор или открывает существующий. Именованный семафор может использоваться для синхронизации выполнения потоков и процессов:

Требования к аргументу *name* приведены в разделе 2.2.

Аргумент *oflag* может принимать значения 0, `O_CREAT`, `O_CREAT | O_EXCL`, как описано в разделе 2.3. Если указано значение `O_CREAT`, третий и четвертый аргументы функции являются обязательными. Аргумент *mode* указывает биты разрешений доступа (табл. 2.3), а *value* указывает начальное значение семафора. Это значение не может превышать константу `SEM_VALUE_MAX`, которая, согласно Posix, должна быть не менее 32767. Бинарные семафоры обычно устанавливаются в 1, тогда как семафоры-счетчики чаще инициализируются большими величинами.

При указании флага `O_CREAT` (без `O_EXCL`) семафор инициализируется только в том случае, если он еще не существует. Если семафор существует, ошибки не возникнет. Ошибка будет возвращена только в том случае, если указаны флаги `O_CREAT | O_EXCL`.

Возвращаемое значение представляет собой указатель на тип `sem_t`. Этот указатель впоследствии передается в качестве аргумента функциям `sem_close`, `sem_wait`, `sem_trywait`, `sem_post` и `sem_getvalue`.

ПРИМЕЧАНИЕ

Кажется странным возвращать `SEM_FAILED` в случае ошибки — нулевой указатель был бы более уместен. В ранних версиях стандарта Posix указывалось возвращаемое значение -1, и во многих реализациях константа `SEM_FAILED` определена как

В Posix.1 мало говорится о битах разрешений, связываемых с семафором при его создании и открытии. Вспомните, мы говорили в связи с табл. 2.2 о том, что для именованных семафоров не нужно даже указывать флаги `O_RDONLY`, `O_WRONLY` и `O_RDWR`. В системах, на которых мы тестируем все программы этой книги (Digital Unix 4.0B и Solaris 2.6), для работы с семафором (его открытия) необходимо иметь к нему доступ как на чтение, так и на запись. Причина, скорее всего, в том, что обе операции, выполняемые с семафором (`post` и `wait`), состоят из считывания текущего значения и последующего его изменения. Отсутствие доступа на чтение или запись в этих реализациях приводит к возвращению функцией `sem_open` ошибки `EACCESS` ("Permission denied").

Открыв семафор с помощью `sem_open`, можно потом закрыть его, вызвав `sem_close`:

Операция закрытия выполняется автоматически при завершении процесса для всех семафоров, которые были им открыты. Автоматическое закрытие осуществляется как при добровольном завершении работы (вызове `exit` или `_exit`), так и при принудительном (с помощью сигнала).

Закрытие семафора не удаляет его из системы. Именованные семафоры Posix обладают по меньшей мере живучестью ядра. Значение семафора сохраняется, даже если ни один процесс не держит его открытым.

Именованный семафор удаляется из системы вызовом `sem_unlink`:

Для каждого семафора ведется подсчет процессов, в которых он является открытим (как и для файлов), и функция `sem_unlink` действует аналогично `unlink` для файлов: объект *name* может быть удален из файловой системы, даже если он открыт какими-либо процессами, но реальное удаление семафора не будет осуществлено до тех пор, пока он не будет окончательно закрыт.

10.3. Функции sem_wait и sem_trywait

Функция `sem_wait` проверяет значение заданного семафора на положительность, уменьшает его на единицу и немедленно возвращает управление процессу. Если значение семафора при вызове функции равно нулю, процесс приостанавливается, до тех пор пока оно снова не станет больше нуля, после чего значение семафора будет уменьшено на единицу и произойдет возврат из функции. Ранее мы отметили, что операция «проверка и уменьшение» должна быть атомарной по отношению к другим потокам, работающим с этим семафором:

Разница между `sem_wait` и `sem_trywait` заключается в том, что последняя не приостанавливает выполнение процесса, если значение семафора равно нулю, а просто немедленно возвращает ошибку `EAGAIN`.

Возврат из функции `sem_wait` может произойти преждевременно, если будет получен сигнал. При этом возвращается ошибка с кодом `EINTR`.

10.4. Функции `sem_post` и `sem_getvalue`

После завершения работы с семафором поток вызывает `sem_post`. Как мы уже говорили в разделе 10.1, этот вызов увеличивает значение семафора на единицу и возобновляет выполнение любых потоков, ожидающих изменения значения семафора:

Функция `sem_getvalue` возвращает текущее значение семафора, помещая его в целочисленную переменную, на которую указывает `valp`. Если семафор заблокирован, возвращается либо 0, либо отрицательное число, модуль которого соответствует количеству потоков, ожидающих разблокирования семафора.

Теперь мы ясно видим отличия семафоров от взаимных исключений и условных переменных. Прежде всего взаимное исключение может быть разблокировано только заблокировавшим его потоком. Для семафоров такого ограничения нет: один из потоков может ожидать изменения значения семафора, чтобы потом уменьшить его с 1 до 0 (действие аналогично блокированию семафора), а другой поток может изменить значение семафора с 0 до 1, что аналогично разблокированию семафора.

Далее, поскольку любой семафор имеет некоторое значение, увеличиваемое операцией `post` и уменьшаемое операцией `wait`, поток может изменить его значение (например, увеличить с 0 до 1), даже если нет потоков, ожидающих его изменения. Если же поток вызывает `pthread_cond_signal` в отсутствие заблокированных при вызове `pthread_cond_wait` потоков, сигнал просто теряется.

Наконец, среди всех функций, работающих со средствами синхронизации — взаимными исключениями, условными переменными, блокировками чтения-записи и семафорами, только одна может быть вызвана из обработчика сигналов: `sem_post`.

ПРИМЕЧАНИЕ

Не следует рассматривать приведенный выше текст как доводы в пользу семафоров. Все средства синхронизации, обсуждаемые в этой книге — взаимные исключения, условные переменные, блокировки чтения-записи, семафоры и блокировка `fcntl`, обладают своими преимуществами и недостатками. Выбирать средства синхронизации для приложения следует с учетом их многочисленных особенностей. Из нашего сравнительного описания можно сделать вывод, что взаимные исключения больше приспособлены для блокировки, условные переменные — для ожидания, а семафоры — для того и другого, и последнее может привести к излишнему усложнению текста программы.

В этом разделе мы напишем несколько простых программ, работающих с именованными семафорами Posix. Эти программы помогут нам узнать особенности функционирования и реализации семафоров. Поскольку именованные семафоры Posix обладают по крайней мере живучестью ядра, для работы с ними мы можем использовать отдельные программы.

Программа semcreate

В листинге 10.3 приведен текст программы, создающей именованный семафор. При вызове программы можно указать параметр -e, обеспечивающий исключающее создание (если семафор уже существует, будет выведено сообщение об ошибке), а параметр -i с числовым аргументом позволяет задать начальное значение семафора, отличное от 1.

22 Поскольку мы всегда указываем флаг O_CREAT, нам приходится вызывать sem_open с четырьмя аргументами. Последние два используются только в том случае, если семафор еще не существует.

23 Мы вызываем sem_close, хотя, если бы мы не сделали этот вызов, семафор все равно закрылся бы автоматически при завершении процесса и ресурсы системы были бы высвобождены.

Программа semunlink

Программа в листинге 10.4 удаляет именованный семафор.

Программа semgetvalue

В листинге 10.5 приведен текст простейшей программы, которая открывает указанный именованный семафор, получает его текущее значение и выводит его.

9 Семафор, который мы открываем, должен быть заранее создан другой программой. Вторым аргументом `sem_open` будет 0: мы не указываем флаг `O_CREAT` и нам не нужно задавать никаких других параметров открытия `0_xxx`.

Программа semwait

Программа в листинге 10.6 открывает именованный семафор, вызывает semwait (которая приостанавливает выполнение процесса, если значение семафора меньше либо равно 0, а при положительном значении семафора уменьшает его на 1), получает и выводит значение семафора, а затем останавливает свою работу навсегда при вызове pause.

Программа sempost

В листинге 10.7 приведена программа, которая выполняет операцию *post* для указанного семафора (то есть увеличивает его значение на 1), а затем получает значение этого семафора и выводит его.

Примеры

Для начала мы создадим именованный семафор в Digital Unix 4.0B и выведем его значение, устанавливаемое по умолчанию при инициализации:

Аналогично очередям сообщений Posix система создает файл семафора с тем именем, которое мы указали при вызове функции.

Теперь подождем изменения семафора и прервем работу программы, установившей блокировку:

Приведенный пример иллюстрирует упомянутые ранее особенности. Во-первых, значение семафора обладает живучестью ядра. Значение 1, установленное при создании семафора, хранится в ядре даже тогда, когда ни одна программа не пользуется этим семафором. Во-вторых, при выходе из программы semwait, заблокировавшей семафор, значение его не изменяется, то есть ресурс остается заблокированным. Это отличает семафоры от блокировок fcntl, описанных в главе 9, которые снимались автоматически при завершении работы процесса.

Покажем теперь, что в этой реализации отрицательное значение семафора используется для хранения информации о количестве процессов, ожидающих разблокирования семафора:

При первом вызове sem_post значение семафора изменилось с -2 на -1 и один из процессов, ожидавших изменения значения семафора, был разблокирован.

Выполним те же действия в Solaris 2.6, обращая внимание на различия в реализации:

Аналогично очередям сообщений Posix файлы создаются в каталоге /tmp, причем указываемое при вызове имя становится суффиксом имен файлов. Разрешения первого файла соответствуют указанным в вызове sem_open, а второй файл, как можно предположить, используется для блокировки доступа.

Проверим, что ядро не осуществляет автоматического увеличения значения семафора при завершении работы процесса, установившего блокировку:

Посмотрим теперь, как меняется значение семафора в этой реализации при появлении новых процессов, ожидающих изменения значения семафора:

Можно заметить отличие по сравнению с результатами выполнения той же последовательности команд в Digital Unix 4.0B: после изменения значения семафора управление сразу же передается ожидающему изменения семафора процессу.

В разделе 7.3 мы описали суть задачи производителей и потребителей и привели несколько возможных ее решений, в которых несколько потоков-производителей заполняли массив, который обрабатывался одним потоком-потребителем.

1. В нашем первом варианте решения (раздел 7.2) потребитель запускался только после завершения работы производителей, поэтому мы могли решить проблему синхронизации, используя единственное взаимное исключение для синхронизации производителей.

2. В следующем варианте решения (раздел 7.5) потребитель запускался до завершения работы производителей, поэтому требовалось использование взаимного исключения (для синхронизации производителей) вместе с условной переменной и еще одним взаимным исключением (для синхронизации потребителя с производителями).

Расширим постановку задачи производителей и потребителей, используя общий буфер в качестве циклического: заполнив последнее поле, производитель (`buff[NBUFF-1]`) возвращается к его началу и заполняет первое поле (`buff[0]`), и потребитель действует таким же образом. Возникает еще одно требование к синхронизации: потребитель не должен опережать производителя. Мы все еще предполагаем, что производитель и потребитель представляют собой отдельные потоки одного процесса, но они также могут быть и просто отдельными процессами, если мы сможем создать для них общий буфер (например, используя разделяемую память, часть 4).

При использовании общего буфера в качестве циклического код должен удовлетворять трем требованиям:

1. Потребитель не должен пытаться извлечь объект из буфера, если буфер пуст.
2. Производитель не должен пытаться поместить объект в буфер, если последний полон.
3. Состояние буфера может описываться общими переменными (индексами, счетчиками, указателями связных списков и т.д.), поэтому все операции с буфером, совершаемые потребителями и производителями, должны быть защищены от потенциально возможной ситуации гонок.

Наше решение использует три семафора:

1. Бинарный семафор с именем `mutex` защищает критические области кода: помещение данных в буфер (для производителя) и изъятие данных из буфера (для потребителя). Бинарный семафор, используемый в качестве взаимного исключения, инициализируется единицей. (Конечно, мы могли бы воспользоваться и обычным взаимным исключением вместо двоичного семафора. См. упражнение 10.10.)
2. Семафор-счетчик с именем `nempty` подсчитывает количество свободных полей в буфере. Он инициализируется значением, равным объему буфера (`NBUFF`).
3. Семафор-счетчик с именем `nstored` подсчитывает количество заполненных полей в буфере. Он инициализируется нулем, поскольку изначально буфер пуст.



Рис. 10.7. Состояние буфера и двух семафоров-счетчиков после инициализации

На рис. 10.7 показано состояние буфера и двух семафоров-счетчиков после завершения инициализации. Неиспользуемые элементы массива выделены темным.

В нашем примере производитель помещает в буфер целые числа от 0 до NLOOP-1 (`buff[0] = 0, buff[1] = 1`), работая с ним как с циклическим. Потребитель считывает эти числа и проверяет их правильность, выводя сообщения об ошибках в стандартный поток вывода.

На рис. 10.8 изображено состояние буфера и семафоров-счетчиков после помещения в буфер трех элементов, но до изъятия их потребителем.



Рис. 10.8. Буфер и семафоры после помещения в буфер трех элементов

Предположим, что потребитель изъял один элемент из буфера. Новое состояние изображено на рис. 10.9.



Рис. 10.9. Буфер и семафоры после удаления первого элемента из буфера

В листинге 10.8 приведен текст функции main, которая создает три семафора, запускает два потока, ожидает их завершения и удаляет семафоры.

6-10 Потоки совместно используют буфер, содержащий NBUFF элементов, и три указателя на семафоры. Как говорилось в главе 7, мы объединяем эти данные в структуру, чтобы подчеркнуть, что семафоры используются для синхронизации доступа к буферу.

19-25 Мы создаем три семафора, передавая их имена функции px_ipc_name. Флаг O_EXCL мы указываем, для того чтобы гарантировать инициализацию каждого семафора правильным значением. Если после преждевременно завершенного предыдущего запуска программы остались неудаленные семафоры, мы обрабатываем эту ситуацию, вызвав перед их созданием sem_unlink и игнорируя ошибки. Мы могли бы проверять возвращение ошибки EEXIST при вызове sem_open с флагом O_EXCL, а затем вызывать sem_unlink и еще раз sem_open, но это усложнило бы программу. Если нам нужно проверить, что запущен только один экземпляр программы (что следует сделать перед созданием семафоров), можно обратиться к разделу 9.7, где описаны методы решения этой задачи.

26-29 Создаются два потока, один из которых является производителем, а другой — потребителем. При запуске никакие аргументы им не передаются.

30-36 Главный поток ждет завершения работы производителя и потребителя, а затем удаляет три семафора.

ПРИМЕЧАНИЕ

Мы могли бы вызвать для каждого семафора sem_close, но это делается автоматически при завершении процесса. А вот удалить имя семафора из файловой системы необходимо явно.

В листинге 10.9 приведен текст функций produce и consume.

44 Производитель вызывает sem_wait для семафора nempty, ожидая появления свободного места. В первый раз при выполнении этой команды значение семафора nempty уменьшится с NBUFF до NBUFF-1.

45-48 Перед помещением нового элемента в буфер производитель должен установить блокировку на семафор mutex. В нашем примере, где производитель просто сохраняет значение в элементе массива с индексом $i \% \text{NBUFF}$, для описания состояния буфера не используется никаких разделяемых переменных (то есть мы не используем связанный список, который нужно было бы обновлять каждый раз при помещении элемента в буфер). Следовательно, установка и снятие семафора mutex не являются обязательными. Тем не менее мы иллюстрируем эту технику, потому что обычно ее применение является необходимым в задачах такого рода (обновление буфера, разделяемого несколькими потоками).

После помещения элемента в буфер блокировка с семафора mutex снимается (его значение увеличивается с 0 до 1) и увеличивается значение семафора nstored. Первый раз при выполнении этой команды значение nstored изменится с начального значения 0 до 1.

57-62 Если значение семафора nstored больше 0, в буфере имеются объекты для обработки. Потребитель изымает один элемент из буфера и проверяет правильность его значения, защищая буфер в момент доступа к нему с помощью семафора mutex. Затем потребитель увеличивает значение семафора nempty, указывая производителю на наличие свободных полей.

Зависание

Что произойдет, если мы по ошибке поменяем местами вызовы `Sem_wait` в функции `consumer` (листинг 10.9)? Предположим, что первым запускается производитель (как в решении, предложенном для упражнения 10.1). Он помещает в буфер NBUFF элементов, уменьшая значение семафора `nempty` от NBUFF до 0 и увеличивая значение семафора `nstored` от 0 до NBUFF. Затем производитель блокируется в вызове `Sem_wait(shared, nempty)`, поскольку буфер полон и помещать элементы больше некуда.

Запускается потребитель и проверяет первые NBUFF элементов буфера. Это уменьшает значение семафора `nstored` от NBUFF до 0 и увеличивает значение семафора `nempty` от 0 до NBUFF. Затем потребитель блокируется в вызове `Sem_wait(shared, nstored)` после вызова `Sem_wait(shared, mutex)`. Производитель мог бы продолжать работу, поскольку значение семафора `nempty` уже отлично от 0, но он вызвал `Sem_wait(shared, mutex)` и его выполнение было приостановлено.

Это называется зависанием программы (deadlock). Производитель ожидает освобождения семафора `mutex`, а потребитель не снимает с него блокировку, ожидая освобождения семафора `nstored`. Но производитель не может изменить `nstored`, пока он не получит семафор `mutex`. Это одна из проблем, которые часто возникают с семафорами: если в программе сделать ошибку, она будет работать неправильно.

ПРИМЕЧАНИЕ

Стандарт Posix позволяет функции `sem_wait` обнаруживать зависание и возвращать ошибку `EDEADLK`, но ни одна из систем, использовавшихся для написания примеров (Digital Unix 4.0B и Solaris 2.6), не обнаружила ошибку в данном случае.

10.7. Блокирование файлов

Вернемся к задаче о порядковом номере из главы 9. Здесь мы напишем новые версии функций `my_lock` и `my_unlock`, использующие именованные семафоры Posix. В листинге 10.10 приведен текст этих функций.

Один из семафоров используется для рекомендательной блокировки доступа к файлу и инициализируется единицей при первом вызове функции. Для получения блокировки мы вызываем `sem_wait`, а для ее снятия — `sem_post`.

До сих пор мы имели дело только с именованными семафорами Posix. Как мы уже говорили, они идентифицируются аргументом *name*, обычно представляющим собой имя файла в файловой системе. Стандарт Posix описывает также семафоры, размещаемые в памяти, память под которые выделяет приложение (тип *sem_t*), а инициализируются они системой:

Размещаемый в памяти семафор инициализируется вызовом *sem_init*. Аргумент *sem* указывает на переменную типа *sem_t*, место под которую должно быть выделено приложением. Если аргумент *shared* равен 0, семафор используется потоками одного процесса, в противном случае доступ к нему могут иметь несколько процессов. Если аргумент *shared* ненулевой, семафор должен быть размещен в одном из видов разделяемой памяти и должен быть доступен всем процессам, использующим его. Как и в вызове *sem_open*, аргумент *value* задает начальное значение семафора.

После завершения работы с размещаемым в памяти семафором его можно уничтожить, вызвав *sem_destroy*.

ПРИМЕЧАНИЕ 1

Функции *sem_open* не требуется параметр, аналогичный *shared*; не требуется ей и атрибут, аналогичный *PTHREAD_PROCESS_SHARED* (упоминавшийся в связи с взаимными исключениями и условными переменными в главе 7), поскольку именованный семафор *всегда* используется совместно несколькими процессами.

ПРИМЕЧАНИЕ 2

Обратите внимание, что для размещаемого в памяти семафора нет ничего аналогичного флагу *O_CREAT*: функция *sem_init* всегда инициализирует значение семафора. Следовательно, нужно быть внимательным, чтобы вызывать *sem_init* только один раз для каждого семафора. (Упражнение 10.2 иллюстрирует разницу в этом смысле между именованным и размещаемым в памяти семафорами.) При вызове *sem_init* для уже инициализированного семафора результат непредсказуем.

ПРИМЕЧАНИЕ 3

Удостоверьтесь, что вы понимаете фундаментальную разницу между *sem_open* и *sem_init*. Первая возвращает указатель на переменную типа *sem_t*, причем выделение места под переменную и ее инициализация выполняются этой же функцией. Напротив, первый аргумент *sem_init* представляет собой указатель на переменную типа *sem_t*, место под которую должен был заранее выделить вызывающий. Функция *sem_init* только инициализирует эту переменную.

ПРИМЕЧАНИЕ 4

Стандарт Posix.1 предупреждает, что для обращения к размещаемым в памяти семафорам можно использовать только указатель, являющийся аргументом при вызове *sem_init*. Использование копий этого указателя может привести к неопределенным результатам.

Функция *sem_init* возвращает -1 в случае ошибки, но она не возвращает 0 в случае успешного завершения. Это действительно странно, и примечание в Обосновании Posix. 1 говорит, что в будущих версиях функция, возможно, начнет возвращать 0 в случае успешного завершения.

Размещаемый в памяти семафор может быть использован в тех случаях, когда нет необходимости использовать имя, связываемое с именованным семафором. Именованные семафоры обычно используются для синхронизации работы неродственных процессов. Имя в этом случае используется для идентификации семафора.

В связи с табл. 1.1 мы говорили о том, что семафоры, размещаемые в памяти, обладают живучестью процесса, но на самом деле их живучесть зависит от типа используемой разделяемой памяти.

Размещаемый в памяти семафор не утрачивает функциональности до тех пор, пока память, в которой он размещен, еще доступна какому-либо процессу.

■ Если размещаемый в памяти семафор совместно используется потоками одного процесса (аргумент *shared* при вызове *sem_init* равен 0), семафор обладает живучестью процесса и удаляется при завершении последнего.

■ Если размещаемый в памяти семафор совместно используется несколькими процессами (аргумент *shared* при вызове *sem_init* равен 1), он должен располагаться в разделяемой памяти, и в этом случае семафор существует столько, сколько существует эта область памяти. Вспомните, что и разделяемая память Posix, и разделяемая память System V обладают живучестью ядра (табл. 1.1). Это значит, что сервер может создать область разделяемой памяти, инициализировать в ней размещаемый в памяти семафор Posix, а затем завершить работу. Некоторое время спустя один или несколько клиентов могут присоединить эту область к своему адресному пространству и получить доступ к хранящемуся в ней семафору.

Предупреждаем, что нижеследующий код не работает так, как ожидается:

Проблема тут в том, что семафор не располагается в разделяемой памяти (см. раздел 10.12). Память, как правило, не делится между дочерним и родительским процессами при вызове `fork`. Дочерний процесс запускается с копией памяти родителя, но это не то же самое, что разделяемая память.

Пример

В качестве иллюстрации перепишем наш пример решения задачи производителей и потребителей из листингов 10.8 и 10.9 для использования размещаемых в памяти семафоров Posix. В листинге 10.11 приведен текст новой программы.

6 Мы объявляем три семафора типа `sem_t`, и теперь это сами семафоры, а не указатели на них.

16-27 Мы вызываем `sem_init` вместо `sem_open*` а затем `sem_destroy` вместо `sem_unlink`. Вызывать `sem_destroy` на самом деле не требуется, поскольку программа все равно завершается.

Остальные изменения обеспечивают передачу указателей на три семафора при вызовах `sem_wait` и `sem_post`.

10.9. Несколько производителей, один потребитель

Решение в разделе 10.6 относится к классической задаче с одним производителем и одним потребителем. Новая, интересная модификация программы позволит нескольким производителям работать с одним потребителем. Начнем с решения из листинга 10.11, в котором использовались размещаемые в памяти семафоры. В листинге 10.12 приведены объявления глобальных переменных и функция main.

4 Глобальная переменная nitems хранит число элементов, которые должны быть совместно произведены. Переменная producers хранит число потоков-производителей. Оба эти значения устанавливаются с помощью аргументов командной строки.

5-10 В структуру shared добавляются два новых элемента: прит, обозначающий индекс следующего элемента, куда должен быть помещен объект (по модулю BUFF), и притвал — следующее значение, которое будет помещено в буфер. Эти две переменные взяты из нашего решения в листингах 7.1 и 7.2. Они нужны для синхронизации нескольких потоков-производителей.

17-20 Два новых аргумента командной строки указывают полное количество элементов, которые должны быть помещены в буфер, и количество потоков-производителей.

21-41 Инициализируем семафоры и запускаем потоки-производители и поток-потребитель. Затем ожидается завершение работы потоков. Эта часть кода практически идентична листингу 7.1.

В листинге 10.13 приведен текст функции produce, которая выполняется каждым потоком-производителем.

49-53 Отличие от листинга 10.8 в том, что цикл завершается, когда nitems объектов будет помещено в буфер всеми потоками. Обратите внимание, что потоки-производители могут получить семафор nempty в любой момент, но только один производитель может иметь семафор mutex. Это защищает переменные прит и nval от одновременного изменения несколькими производителями.

50-51 Нам нужно аккуратно обработать завершение потоков-производителей. После того как последний объект помещен в буфер, каждый поток выполняет

в начале цикла, что уменьшает значение семафора nempty. Но прежде, чем поток будет завершен, он должен увеличить значение этого семафора, потому что он не помещает объект в буфер в последнем проходе цикла. Завершающий работу поток должен также освободить семафор mutex, чтобы другие производители смогли продолжить функционирование. Если мы не увеличим семафор nempty по завершении процесса и если производителей будет больше, чем мест в буфере, лишние потоки будут заблокированы навсегда, ожидая освобождения семафора nempty, и никогда не завершат свою работу.

Функция consume в листинге 10.14 проверяет правильность всех записей в буфере, выводя сообщение при обнаружении ошибки.

Условие завершения единственного потока-потребителя звучит просто: он считает все потребленные объекты и останавливается по достижении nitems.

10.10. Несколько производителей, несколько потребителей

Следующее изменение, которое мы внесем в нашу программу, будет заключаться в добавлении возможности одновременной работы нескольких потребителей вместе с несколькими производителями. Есть ли смысл в наличии нескольких потребителей — зависит от приложения. Автор видел два примера, в которых использовался этот метод.

1. Программа преобразования IP-адресов в имена узлов. Каждый потребитель берет IP-адрес, вызывает `gethostbyaddr` (раздел 9.6 [24]), затем дописывает имя узла к файлу. Поскольку каждый вызов `gethostbyaddr` обрабатывается неопределенное время, порядок IP-адресов в буфере будет, скорее всего, отличаться от порядка имен узлов в файле, созданном потоками-потребителями. Преимущество этой схемы в параллельности выполнения вызовов `gethostbyaddr` (каждый из которых может работать несколько секунд) — по одному на каждый поток-потребитель.

ПРИМЕЧАНИЕ

Предполагается наличие версии `gethostbyaddr`, допускающей многократное вхождение, что не всегда верно. Если эта версия недоступна, можно хранить буфер в разделяемой памяти и использовать процессы вместо потоков.

2. Программа, принимающая дейтаграммы UDP, обрабатывающая их и записывающая результат в базу данных. Каждая дейтаграмма обрабатывается одним потоком-потребителем, которые выполняются параллельно для ускорения процесса. Хотя дейтаграммы записываются в базу данных в порядке, вообще говоря, отличном от порядка их приема,строенная схема упорядочения записей в базе данных справляется с этой проблемой.

В листинге 10.15 приведены глобальные переменные программы.

4-12 Количество потоков-потребителей является глобальной переменной, устанавливаемой из командной строки. В структуру `shared` добавилось два новых поля: `nget` — номер следующего объекта, получаемого одним из потоков-потребителей, и `ngetval` — соответствующее значение.

Функция `main`, текст которой приведен в листинге 10.16, запускает несколько потоков-потребителей и потоков-производителей одновременно.

19-23 Новый аргумент командной строки указывает количество потоков-потребителей. Для хранения идентификаторов потоков-потребителей выделяется место под специальный массив (`tid_consume`), а для подсчета обработанных каждым потоком объектов выделяется массив `conscount`.

24-50 Создаются несколько потоков-производителей и потребителей, после чего основной поток ждет их завершения.

Функция `produce` содержит одну новую строку по сравнению с листингом 10.13. В части кода, относящейся к завершению потока-производителя, появляется строка, отмеченная знаком +:

Снова нам нужно быть аккуратными при обработке завершения процессов-производителей и потребителей. После обработки всех объектов в буфере все потребители блокируются в вызове

Производителям приходится увеличивать семафор `nstored` для разблокирования потребителей, чтобы они узнали, что работа завершена. Функция `consume` приведена в листинге 10.17.

79-83 Функция `consume` сравнивает `nget` и `nitems`, чтобы узнать, когда следует остановиться (аналогично функции `produce`). Обработав последний объект в буфере, потоки-потребители блокируются, ожидая изменения семафора `nstored`. Когда завершается очередной поток-потребитель, он увеличивает семафор `nstored`, давая возможность завершить работу другому потоку-потребителю.

10.11. Несколько буферов

Во многих программах, обрабатывающих какие-либо данные, можно встретить цикл вида

Например, программы, обрабатывающие текстовые файлы, считывают строку из входного файла, выполняют с ней некоторые действия, а затем записывают строку в выходной файл. Для текстовых файлов вызовы `read` и `write` часто заменяются на функции стандартной библиотеки ввода-вывода `fgets` и `fputs`.

На рис. 10.11 изображена иллюстрация к такой схеме. Здесь функция `reader` считывает данные из входного файла, а функция `writer` записывает данные в выходной файл. Используется один буфер.



Рис. 10.10. Процесс считывает данные в буфер, а потом записывает его содержимое в другой файл



Рис. 10.11. Один процесс, считающий данные в буфер и записывающий их в файл

На рис. 10.10 приведена временная диаграмма работы такой программы. Числа слева проставлены в условных единицах времени. Предполагается, что операция чтения занимает 5 единиц, записи — 7, а обработка данных между считыванием и записью требует 2 единицы времени.

Можно изменить это приложение, разделив процесс на отдельные потоки, как показано на рис. 10.12. Здесь используется два потока (а не процесса), поскольку глобальный буфер автоматически разделяется между ними. Мы могли бы разделить приложение и на два процесса, но это потребовало бы использования разделяемой памяти, с которой мы еще не знакомы.



Рис. 10.12. Разделение копирования файла между двумя потоками

Разделение операций между потоками (или процессами) требует использования какой-либо формы уведомления между ними. Считывающий поток должен уведомлять записывающий о готовности буфера к операции записи, а записывающий должен уведомлять считающий о том, что буфер пуст и его можно заполнять снова. На рис. 10.13 изображена временная диаграмма для новой схемы.



Рис. 10.13. Копирование файла двумя потоками

Предполагается, что для обработки данных в буфере требуется две единицы времени. Важно отметить, что разделение чтения и записи между двумя потоками ничуть не ускорило выполнение операции копирования в целом. Мы не выиграли в скорости, мы просто распределили выполнение задачи между двумя потоками (или процессами).

В этих диаграммах мы игнорируем множество тонкостей. Например, большая часть ядер Unix выявляет операцию последовательного считывания файла и осуществляет асинхронное упреждающее чтение следующего блока данных еще до поступления запроса. Это может ускорить работу процесса, считающего данные. Мы также игнорируем влияние других процессов на наши считающий и записывающий потоки, а также влияние алгоритмов разделения времени, реализованных в ядре.

Следующим шагом будет использование двух потоков (или процессов) и двух буферов. Это называется классическим решением с двойной буферизацией; схема его изображена на рис. 10.14.



Рис. 10.14. Копирование файла двумя потоками с двумя буферами

На нашем рисунке считающий поток помещает данные в первый буфер, а записывающий берет их из второго. После этого потоки меняются местами.

На рис. 10.15 изображена временная диаграмма процесса с двойной буферизацией. Считывающий поток помещает данные в буфер № 1, а затем уведомляет записывающий о том, что буфер готов к обработке. Затем считающий процесс помещает данные в буфер № 2, а записывающий берет их из буфера № 1.

В любом случае, мы ограничены скоростью выполнения самой медленной операции — операции записи. После выполнения первых двух операций считывания серверу приходится ждать две дополнительные единицы времени, составляющие разницу в скорости выполнения операций чтения и записи. Тем не менее для нашего гипотетического примера полное время работы будет сокращено почти вдвое.

Обратите внимание, что операции записи выполняются так быстро, как только возможно. Они разделены промежутками времени всего лишь в 2 единицы, тогда как в предыдущих примерах между ними проходило 9 единиц времени (рис. 10.10 и 10.13). Это может оказаться выгодным при работе с некоторыми устройствами типа накопителей на магнитной ленте, которые функционируют быстрее, если данные записываются с максимальной возможной скоростью (это называется потоковым режимом — *streaming mode*).



Рис. 10.15. Процесс с двойной буферизацией

Интересно, что задача с двойной буферизацией представляет собой лишь частный случай общей задачи производителей и потребителей.

Изменим нашу программу так, чтобы использовать несколько буферов. Начнем с решения из листинга 10.11, в котором использовались размещаемые в памяти семафоры. Мы получим даже не двойную буферизацию, а работу с произвольным числом буферов (задается NBUFF). В листинге 10.18 даны глобальные переменные и функция main.

2-9 Структура `shared` содержит массив структур `buff`, которые состоят из буфера и его счетчика. Мы создаем NBUFF таких буферов.

18 Аргумент командной строки интерпретируется как имя файла, который копируется в стандартный поток вывода.

В листинге 10.19 приведен текст функций `produce` и `consume`.

40-42 Критическая область, защищаемая семафором `mutex`, в данном примере пуста. Если бы буферы данных представляли собой связный список, здесь мы могли бы удалять буфер из списка, не конфликтую при этом с производителем. Но в нашем примере, где мы просто переходим к следующему буферу с единственным потоком-производителем, защищать нам просто нечего. Тем не менее мы оставляем операции установки и снятия блокировки, подчеркивая, что они могут потребоваться в новых версиях кода.

43-49 Каждый раз, когда производитель получает пустой буфер, он вызывает функцию `read`. При возвращении из `read` увеличивается семафор `nstored`, уведомляя потребителя о том, что буфер готов. При возвращении функцией `read` значения 0 (конец файла) семафор увеличивается, а производитель завершает работу.

57-68 Поток-потребитель записывает содержимое буферов в стандартный поток вывода. Буфер, содержащий нулевой объем данных, обозначает конец файла. Как и в потоке-производителе, критическая область, защищенная семафором `mutex`, пуста.

ПРИМЕЧАНИЕ

В разделе 22.3 книги [24] мы разработали пример с несколькими буферами. В этом примере производителем был обработчик сигнала `SIGIO`, а потребитель представлял собой основной цикл обработки (функцию `dg_echo`). Разделяемой переменной был счетчик `pniecie`. Потребитель блокировал сигнал `SIGIO` на время проверки или изменения счетчика.

10.12. Использование семафоров несколькими процессами

Правила совместного использования размещаемых в памяти семафоров несколькими процессами просты: сам семафор (переменная типа `sem_t`, адрес которой является первым аргументом `sem_init`) должен находиться в памяти, разделяемой всеми процессами, которые хотят его использовать, а второй аргумент функции `sem_init` должен быть равен 1.

ПРИМЕЧАНИЕ

Эти правила аналогичны требованиям к разделению взаимного исключения, условной переменной или блокировки чтения-записи между процессами: средство синхронизации (переменная типа `pthread_mutex_t`, `pthread_cond_t` или `pthread_rwlock_t`) должно находиться в разделяемой памяти и инициализироваться с атрибутом `PTHREAD_PROCESS_SHARED`.

Что касается именованных семафоров, процессы всегда могут обратиться к одному и тому же семафору, указав одинаковое имя при вызове `sem_open`. Хотя указатели, возвращаемые `sem_open` отдельным процессам, могут быть различны, все функции, работающие с семафорами, будут обращаться к одному и тому же именованному семафору.

Что произойдет, если мы вызовем функцию `sem_open`, возвращающую указатель на тип `sem_t`, а затем вызовем `fork?` В описании функции `fork` в стандарте Posix.1 говорится, что «все открытые родительским процессом семафоры будут открыты и в дочернем процессе». Это означает, что нижеследующий код верен:

ПРИМЕЧАНИЕ

Причина, по которой следует аккуратно относиться к передаче семафоров при порождении процессов, заключается в том, что состояние семафора может храниться в переменной типа `sem_t`, но для его работы может требоваться и другая информация (например, дескрипторы файлов). В следующей главе мы увидим, что семафоры System V однозначно определяются их целочисленными идентификаторами, возвращаемыми функцией `semget`. Любой процесс, которому известен идентификатор, может получить доступ к семафору. Вся информация о семафоре System V хранится в ядре, а целочисленный идентификатор просто указывает номер семафора ядру.

Стандартом Posix определены два ограничения на семафоры:

- SEM_NSEMS_MAX — максимальное количество одновременно открытых семафоров для одного процесса (Posix требует, чтобы это значение было не менее 256);
- SEM_VALUE_MAX — максимальное значение семафора (Posix требует, чтобы оно было не меньше 32767).

Две эти константы обычно определены в заголовочном файле `<unistd.h>` и могут быть получены во время выполнения вызовом `sysconf`, как мы показываем ниже.

Пример: программа semsysconf

Программа в листинге 10.20 вызывает sysconf и выводит два ограничения на семафоры, зависящие от конкретной реализации.

При запуске этой программы в наших двух тестовых системах получим следующий результат:

Займемся реализацией именованных семафоров Posix с помощью каналов FIFO. Именованный семафор реализуется как канал FIFO с конкретным именем. Неотрицательное количество байтов в канале соответствует текущему значению семафора. Функция `sem_post` помещает 1 байт в канал, а `sem_wait` считывает его оттуда (приостанавливая выполнение процесса, если канал пуст, а именно этого мы и хотим). Функция `sem_open` создает канал FIFO, если указан флаг `O_CREAT`; открывает его дважды (один раз на запись, другой — на чтение) и при создании нового канала FIFO помещает в него некоторое количество байтов, указанное в качестве начального значения.

ПРИМЕЧАНИЕ

Этот и последующие разделы данной главы содержат усложненный материал, который можно при первом чтении пропустить.

Приведем текст нашего заголовочного файла `semaphore.h`, определяющего фундаментальный тип `sem_t` (листинг 10.21).

1-5 Новая структура данных содержит два дескриптора, один из которых предназначен для чтения из FIFO, а другой — для записи. Для единообразия мы храним оба дескриптора в массиве из двух элементов, в котором первый дескриптор всегда открыт на чтение, а второй — на запись.

Поле `sem_magic` содержит значение `SEM_MAGIC`, если структура проинициализирована. Это значение проверяется всеми функциями, которым передается указатель на тип `sem_t`, чтобы гарантировать, что передан был действительно указатель на заранее инициализированную структуру, а не на произвольную область памяти. При закрытии семафора этому полю присваивается значение 0. Этот метод хотя и не совершенен, но дает возможность обнаружить некоторые ошибки при написании программ.

Функция sem_open

В листинге 10.22 приведен текст функции `sem_open`, которая создает новый семафор или открывает существующий.

13-17 Если при вызове указан флаг `O_CREAT`, должно быть указано четыре аргумента, а не два. Мы вызываем `va_start`, после чего переменная `ap` указывает на последний явно указанный аргумент (`oflag`). Затем мы используем `ap` и функцию `va_arg` для получения значений третьего и четвертого аргументов. Работу со списком аргументов переменной длины и использование нашего типа `va_mode_t` мы обсуждали в связи с листингом 5.17.

18-23 Создается новый канал FIFO, имя которого было указано при вызове функции. Как мы отмечали в разделе 4.6, эта функция возвращает ошибку `EEXIST`, если канал уже существует. Если при вызове `sem_open` флаг `O_EXCL` не был указан, мы пропускаем эту ошибку; но нам не нужно будет инициализировать этот канал, так что мы при этом сбрасываем флаг `O_CREAT`.

25-37 Мы выделяем место для типа `sem_t`, который содержит два дескриптора. Затем мы дважды открываем канал FIFO: один раз только на чтение, а другой — только на запись. При этом мы не хотим блокирования при вызове `open`, поэтому указываем флаги `O_NONBLOCK` при открытии очереди только для чтения (вспомните табл. 4.1). Мы также указываем флаг `O_NONBLOCK` при открытии канала на запись, но это предназначено для обнаружения переполнения (на тот случай, если мы попытаемся записать больше, чем позволяет `PIPE_BUF`). После открытия канала мы отключаем неблокируемый режим для дескриптора, открытого на чтение.

38-42 Если мы создали семафор, его нужно проинициализировать, записав в канал FIFO `value` байтов. Если указанное при вызове значение `value` превышает определенное реализацией ограничение `PIPE_BUF`, вызов `write` после переполнения FIFO вернет ошибку с кодом `EAGAIN`.

Функция sem_close

Текст функции `sem_close` приведен в листинге 10.23.

11-15 Мы закрываем оба дескриптора и освобождаем память, выделенную под тип `sem_t`.

Функция sem_unlink

Функция `sem_unlink`, текст которой приведен в листинге 10.24, удаляет из файловой системы наш семафор. Она просто вызывает `unlink`.

Функция sem_post

В листинге 10.25 приведен текст функции `sem_post`, которая увеличивает значение семафора.

11-12 Мы записываем один байт в FIFO. Если канал был пуст, это приведет к возобновлению выполнения всех процессов, заблокированных в вызове `read` для этого канала.

Функция sem_wait

Последняя функция для работы с именованными семафорами Posix — `sem_wait`. Ее текст приведен в листинге 10.26.

11-12 Мы считываем 1 байт из канала FIFO, причем работа приостанавливается, если канал пуст.

Мы еще не реализовали функцию `sem_trywait`, но это можно сделать, установив флаг отключения блокировки для канала и используя обычный вызов `read`. Мы также не реализовали функцию `sem_getvalue`. В некоторых реализациях при вызове функции `stat` или `fstat` возвращается количество байтов в именованном или неименованном канале, причем оно помещается в поле `st_size` структуры `stat`. Однако это не гарантируется стандартом Posix и, следовательно, не обязательно будет работать в других системах. Пример реализации этих двух функций для работы с семафорами Posix приведен в следующем разделе.

Теперь займемся реализацией именованных семафоров Posix с помощью отображаемых в память файлов вместе со взаимными исключениями и условными переменными Posix. Реализация, аналогичная данной, приведена в разделе B.11.3 Обоснования стандарта IEEE 1996 [8].

ПРИМЕЧАНИЕ

Отображаемые в память файлы описаны в главах 12 и 13. Данный раздел можно отложить, с тем чтобы вернуться к нему после прочтения этих глав.

Прежде всего приведем текст нашего заголовочного файла `semaphore.h` (листинг 10.27), в котором определяется фундаментальный тип `sem_t`.

1-7 Структура данных семафора содержит взаимное исключение, условную переменную и беззнаковое целое, в котором хранится текущее значение семафора. Как уже говорилось в связи с листингом 10.21, поле `sem_magic` получает значение `SEM_MAGIC` при инициализации структуры.

Функция sem_open

В листинге 10.28 приведен текст первой части функции `sem_open`, которая может использоваться для создания нового семафора или открытия существующего.

19-23 Если при вызове функции указан флаг `O_CREAT`, мы должны принять четыре аргумента, а не два. Работа со списком аргументов переменной длины с помощью типа `va_mode_t` уже обсуждалась в связи с листингом 5.17, где мы использовали метод, аналогичный примененному здесь. Мы сбрасываем бит `user-execute` переменной `mode` (`S_IXUSR`) по причинам, которые вскоре будут раскрыты. Создается файл с указанным именем, и для него устанавливается бит `user-execute`.

24-32 Если бы при указании флага `O_CREAT` мы просто открывали файл, отображали в память его содержимое и инициализировали поля структуры `sem_t`, у нас возникла бы ситуация гонок. Эта ситуация также уже обсуждалась в связи с листингом 5.17, и там мы воспользовались тем же методом, что и сейчас. Такая же ситуация гонок встретится нам, когда мы будем разбираться с листингом 10.37.

33-37 Мы устанавливаем размер созданного файла, записывая в него заполненную нулями структуру. Поскольку мы знаем, что только что созданный файл имеет размер 0, для установки его размера мы вызываем именно `write`, но не `ftruncate`, потому что, как мы отмечаем в разделе 13.3, Posix не гарантирует, что `ftruncate` срабатывает при увеличении размера обычных файлов.

38-42 Файл отображается в память вызовом `mmap`. Этот файл будет содержать текущее значение структуры типа `sem_t`, хотя, поскольку мы только что отобразили файл в память, мы обращаемся к нему через указатель, возвращаемый `mmap`, и никогда не вызываем `read` или `write`.

43-57 Мы инициализируем три поля структуры `sem_t`: взаимное исключение, условную переменную и значение семафора. Поскольку именованный семафор Posix может совместно использоваться всеми процессами с соответствующими правами, которым известно его имя, при инициализации взаимного исключения и условной переменной необходимо указать атрибут `PTHREAD_PROCESS_SHARED`. Чтобы осуществить это для взаимного исключения, нужно сначала проинициализировать атрибуты, вызвав `pthread_mutexattr_init`, затем установить атрибут совместного использования потоками, вызвав `pthread_mutexattr_setpshared`, а затем проинициализировать взаимное исключение вызовом `pthread_mutex_init`. Аналогичные действия придется выполнить и для условной переменной. Необходимо аккуратно уничтожать переменные, в которых хранятся атрибуты, при возникновении ошибок.

58-61 Наконец мы помещаем в файл начальное значение семафора. Предварительно мы сравниваем его с максимально разрешенным значением семафора, которое может быть получено вызовом `sysconf` (раздел 10.13).

62-67 После инициализации семафора мы сбрасываем бит `user-execute`. Это указывает на то, что семафор был успешно проинициализирован. Затем мы закрываем файл вызовом `close`, поскольку он уже был отображен в память и нам не нужно держать его открытым.

В листинге 10.29 приведен текст второй половины функции `sem_open`. Здесь возникает ситуация гонок, обрабатываемая так же, как уже обсуждавшаяся в связи с листингом 5.19.

69-78 Здесь мы завершаем нашу работу, если либо не указан флаг `O_CREAT`, либо он указан, но семафор уже существует. В том и в другом случае мы открываем существующий семафор. Мы открываем файл вызовом `open` для чтения и записи, а затем отображаем его содержимое в адресное пространство процесса вызовом `mmap`.

ПРИМЕЧАНИЕ

Теперь легко понять, почему в Posix.1 сказано, что «обращение к копиям семафора приводит к неопределенным результатам». Если именованный семафор реализован через отображение файла в память, он отображается в адресное пространство всех процессов, в которых он открыт. Это осуществляется функцией `sem_open` для каждого процесса в отдельности. Изменения, сделанные одним процессом (например, изменение счетчика семафора), становятся доступны другим процессам через отображение в память. Если мы сделаем свою собственную копию структуры `sem_t`, она уже не будет общей для всех процессов. Хотя нам и может показаться, что вызовы срабатывают (функции для работы с семафором не будут возвращать ошибок, по крайней мере до вызова `sem_close`, которая не сможет отключить отображение для копии отраженного файла), с другими процессами мы при этом взаимодействовать не сможем. Однако заметьте (табл. 1.4), что области памяти с отображаемыми файлами передаются дочерним процессам при вызове `fork`, поэтому создание копии семафора ядром при порождении нового процесса проблем не вызовет.

79-96 Мы должны подождать, пока семафор не будет проинициализирован (если несколько потоков пытаются создать семафор приблизительно одновременно). Для этого мы вызываем stat и проверяем биты разрешений файла (поле st_mode структуры stat). Если бит user-execute снят, структура успешно проинициализирована.

97-108 При возникновении ошибки нужно аккуратно вернуть ее код.

Функция sem_close

В листинге 10.30 приведен текст нашей функции `sem_close`, которая просто вызывает `munmap` для отображенного в память файла. Если вызвавший процесс продолжит пользоваться указателем, который был ранее возвращен `sem_open`, он получит сигнал `SIGSEGV`.

Функция sem_unlink

Текст функции `sem_unlink` приведен в листинге 10.31. Она просто удаляет файл, через который реализован данный семафор, вызывая функцию `unlink`.

Функция sem_post

В листинге 10.32 приведен текст функции `sem_post`, которая увеличивает значение семафора, возобновляя выполнение всех процессов, заблокированных в ожидании этого события.

11-18 Прежде чем работать со структурой, нужно заблокировать соответствующее взаимное исключение. Если значение семафора изменяется с 0 на 1, нужно вызвать `pthread_cond_signal`, чтобы возобновилось выполнение одного из процессов, зарегистрированных на уведомление по данной условной переменной.

Функция sem_wait

В листинге 10.33 приведен текст функции `sem_wait`, которая ожидает изменения значения семафора с 0 на положительное, после чего уменьшает его на 1.

11-18 Прежде чем работать с семафором, нужно заблокировать соответствующее взаимное исключение. Если значение семафора 0, выполнение процесса приостанавливается в вызове `pthread_cond_wait` до тех пор, пока другой процесс не вызовет `pthread_cond_signal` для этого семафора, изменив его значение с 0 на 1. После того как значение становится ненулевым, мы уменьшаем его на 1 и разблокируем взаимное исключение.

Функция sem_trywait

В листинге 10.34 приведен текст функции `sem_trywait`, которая представляет собой просто неблокируемый вариант функции `sem_wait`.

11-22 Мы блокируем взаимное исключение и проверяем значение семафора. Если оно положительно, мы вычитаем из него 1 и возвращаем вызвавшему процессу код 0. В противном случае возвращается -1, а переменной `errno` присваивается код ошибки `EAGAIN`.

Функция sem_getvalue

В листинге 10.35 приведен текст последней функции в этой реализации — `sem_getvalue`. Она возвращает текущее значение семафора.

11-16 Мы блокируем соответствующее взаимное исключение и считываем значение семафора.

Из этой реализации видно, что семафорами пользоваться проще, чем взаимными исключениями и условными переменными.

Приведем еще один пример реализации именованных семафоров Posix — на этот раз с использованием семафоров System V. Поскольку семафоры System V появились раньше, чем семафоры Posix, эта реализация позволяет использовать последние в системах, где их поддержка не предусмотрена производителем.

ПРИМЕЧАНИЕ

Семафоры System V описаны в главе 11. Этот раздел можно пропустить при первом чтении, с тем чтобы вернуться к нему по прочтении 11 главы.

Начнем, как обычно, с заголовочного файла `semaphore.h` (листинг 10.36), который определяет фундаментальный тип данных `sem_t`.

1-5 Мы реализуем именованный семафор Posix с помощью набора семафоров System V, состоящего из одного элемента. Структура данных семафора содержит идентификатор семафора System V и магическое число (обсуждавшееся в связи с листингом 10.21).

Функция sem_open

В листинге 10.37 приведен текст первой половины функции `sem_open`, которая создает новый семафор или открывает существующий.

20-24 Если вызвавший процесс указывает флаг `O_CREAT`, мы знаем, что функции будут переданы четыре аргумента, а не два. Работа со списком аргументов переменной длины и типом данных `va_mode_t` обсуждалась в связи с листингом 5.17.

25-30 Создается обычный файл с именем, указываемым при вызове функции. Это делается для того, чтобы указать его имя при вызове функции `ftok` для последующей идентификации семафора. Аргумент `oflag`, принятый от вызвавшего процесса, передается функции `open` для дополнительного файла, что позволяет создать его, если он еще не существует, и вернуть ошибку `EEXIST`, если файл существует и указан флаг `O_EXCL`. Дескриптор файла затем закрывается, поскольку единственная цель создания файла была в использовании его имени при вызове `ftok`, преобразующей полное имя в ключ System V IPC (раздел 3.2).

32-33 Мы преобразуем константы `O_CREAT` и `O_EXCL` в соответствующие константы System V `IPC_xxx` и вызываем `semget` для создания набора семафоров System V, состоящего из одного элемента. Флаг `IPC_EXCL` указывается всегда, чтобы можно было определить, существовал ли семафор до вызова функции или был создан ею.

34-50 В разделе 11.2 описана фундаментальная проблема, связанная с инициализацией семафоров System V, а в разделе 11.6 приведен код, позволяющий исключить потенциальную ситуацию гонок. Здесь мы пользуемся аналогичным методом. Первый поток, который создает семафор (вспомните, что мы всегда указываем флаг `IPC_EXCL`), инициализирует его значением 0 с помощью команды `SETPVAL` при вызове `semctl`, а затем устанавливает запрошенное вызвавшим процессом начальное значение с помощью `semop`. Мы можем быть уверены, что значение `sem_otime` семафора функцией `semget` устанавливается в 0 и будет изменено на ненулевое вызовом `semop`. Следовательно, любой поток, работающий с существующим семафором, будет знать, что он уже проинициализирован, если значение `sem_otime` будет отлично от 0.

40-44 Мы проверяем начальное значение, указанное вызвавшим процессом, поскольку семафоры System V обычно хранятся как беззнаковые короткие целые (`unsigned short`, структура `sem` в разделе 11.1) с максимальным значением 32767 (раздел 11.7), тогда как семафоры Posix обычно хранятся как целые с максимально возможным размером (раздел 10.13). Константа `SEMVMX` определяется некоторыми реализациями как максимальное значение семафора System V, а если она не определена, то мы определяем ее равной 32 767 в листинге 10.36.

52-53 Если семафор уже существует и вызвавший процесс не указал флаг `O_EXCL`, ошибка не возвращается. В этом случае программа переходит к открытию (не созданию) существующего семафора.

В листинге 10.38 приведен текст второй половины функции `sem_open`.

55-63 Если семафор уже создан (флаг `O_CREAT` не указан или указан, но без `O_EXCL`, а семафор существует), мы открываем семафор System V с помощью `semget`. Обратите внимание, что в вызове `sem_open` указывать аргумент `mode` не нужно, если не указан флаг `O_CREAT`, но вызов `semget` требует указания режима доступа, даже если открывается существующий семафор. Ранее в тексте функции мы присваивали значение по умолчанию (константу `SVSEM_MODE` из нашего заголовочного файла `unpripc.h`) переменной, которую теперь передаем `semget`, если не указан флаг `O_CREAT`.

64-72 Проверяем, что семафор уже инициализирован, вызывая `semctl` с командой `IPC_STAT` и сравнивая значение поля `sem_otime` возвращаемой структуры с нулем.

73-78 Когда возникает ошибка, мы аккуратно вызываем все последующие функции, чтобы не изменить значение `errno`.

79-84 Мы выделяем память под структуру `sem_t` и помещаем в нее идентификатор семафора System V. Функция возвращает указатель на эту структуру.

Функция sem_close

В листинге 10.39 приведен текст функции `sem_close`, которая вызывает `free` для освобождения динамически выделенной под структуру `sem_t` памяти.

Функция sem_unlink

Функция `sem_unlink`, текст которой приведен в листинге 10.40, удаляет вспомогательный файл и семафор System V, связанные с указанным ей семафором Posix.

8-16 Функция `ftok` преобразует полное имя файла в ключ System V IPC. После этого вспомогательный файл удаляется вызовом `unlink` (именно в этом месте кода, на тот случай, если одна из последующих функций вернет ошибку). Затем мы открываем семафор System V вызовом `semget` и удаляем его с помощью команды `IPC_RMID` для `semctl`.

Функция sem_post

В листинге 10.41 приведен текст функции `sem_post`, которая увеличивает значение семафора.

11-16 Мы вызываем `semop` с операцией, увеличивающей значение семафора на 1.

Функция sem_wait

Следующая функция приведена в листинге 10.42; она называется `sem_wait` и ожидает изменения значения семафора с нулевого на ненулевое, после чего уменьшает значение семафора на 1.

11-16 Мы вызываем `semop` с операцией, уменьшающей значение семафора на 1.

Функция sem_trywait

В листинге 10.43 приведен текст нашей функции `sem_trywait`, которая представляет собой неблокирующую версию `sem_wait`.

13 Единственное отличие от функции `sem_wait` из листинга 10.42 заключается в том, что флагу `sem_flg` присваивается значение `IPC_NOWAIT`. Если операция не может быть завершена без блокирования вызвавшего потока, функция `semop` возвращает ошибку `EAGAIN`, а это именно тот код, который должен быть возвращен `sem_trywait`, если операция не может быть завершена без блокирования потока.

Функция sem_getvalue

Последняя функция приведена в листинге 10.44. Это функция `sem_getvalue`, возвращающая текущее значение семафора.

11-14 Текущее значение семафора получается отправкой команды GETVAL функции `semctl`.

10.17. Резюме

Семафоры Posix представляют собой семафоры-счетчики, для которых определены три основные операции:

1. Создание семафора.
2. Ожидание изменения значения семафора на ненулевое и последующее уменьшение значения.
3. Увеличение значения семафора на 1 и возобновление выполнения всех процессов, ожидающих его изменения.

Семафоры Posix могут быть именованными или неименованными (размещаемыми в памяти). Именованные семафоры всегда могут использоваться отдельными процессами, тогда как размещаемые в памяти должны для этого изначально планироваться как разделяемые между процессами. Эти типы семафоров также отличаются друг от друга по живучести: именованные семафоры обладают по меньшей мере живучестью ядра, тогда как размещаемые в памяти обладают живучестью процесса.

Задача производителей и потребителей является классическим примером для иллюстрации использования семафоров. В этой главе первое решение состояло из одного потока-производителя и одного потока-потребителя; второе решение имело нескольких производителей и одного потребителя, а последнее решение допускало одновременную работу и нескольких потребителей. Затем мы показали, что классическая задача двойной буферизации является частным случаем задачи производителей и потребителей с одним производителем и одним потребителем.

В этой главе было приведено три примера возможной реализации семафоров Posix. Первый пример был самым простым, в нем использовались каналы FIFO, а большая часть забот по синхронизации ложилась на ядро (функции read и write). Следующая реализация использовала отображение файлов в память (аналогично реализации очередей сообщений Posix из раздела 5.8), а также взаимное исключение и условную переменную (для синхронизации). Последняя реализация была основана на семафорах System V и представляла собой, по сути, удобный интерфейс для работы с ними.

Упражнения

1. Измените функции `produce` и `consume` из раздела 10.6 следующим образом. Поменяйте порядок двух вызовов `Sem_wait` в потребителе, чтобы возникла ситуация зависания (как описано в разделе 10.6). Затем добавьте вызов `printf` перед каждым `Sem_wait`, чтобы было ясно, какой из потоков ожидает изменения семафора. Добавьте еще один вызов `printf` после каждого `Sem_wait`, чтобы можно было определить, какой поток получил управление. Уменьшите количество буферов до двух, а затем откомпилируйте и выполните эту программу, чтобы убедиться, что она зависнет.

2. Предположим, что запущено четыре экземпляра программы,зывающей функцию `my_lock` из листинга 10.10:

Каждый из четырех процессов запускается с значением `initflag`, равным 0, поэтому при вызове `sem_open` всегда указывается `O_CREAT`. Нормально ли это?

3. Что произойдет в предыдущем примере, если одна из четырех программ будет завершена после вызова `my_lock`, но перед вызовом `my_unlock`?

4. Что произошло бы с программой в листинге 10.22, если бы мы не инициализировали оба дескриптора значением -1?

5. Почему в листинге 10.22 мы сохраняем значение `errno`, а затем восстанавливаем его, вместо того чтобы написать просто:

6. Что произойдет, если два процесса вызовут нашу реализацию `sem_open` через FIFO (листинг 10.22) примерно одновременно, указывая флаг `O_CREAT` и начальное значение 5? Может ли канал быть инициализирован (неправильно) значением 10?

7. В связи с листингами 10.28 и 10.29 мы описали возможную ситуацию гонок в случае, если два процесса пытаются создать семафор примерно одновременно. Однако решение предыдущей задачи в листинге 10.22 не создавало ситуации гонок. Объясните это.

8. Стандарт Posix.1 указывает дополнительную возможность для функции `semwait`: она может прерываться перехватываемым сигналом и возвращать код `EINTR`. Напишите тестовую программу, которая определяла бы, есть ли такая возможность в вашей реализации.

Запустите эту тестовую программу с нашими реализациями, использующими FIFO (раздел 10.14), отображение в память (раздел 10.15) и семафоры System V (раздел 10.16).

9. Какая из трех реализаций `sem_post` этой главы является функцией типа `async-signal-safe` (табл. 5.1)?

10. Измените решение задачи о потребителе и производителе в разделе 10.6 так, чтобы для переменной `mutex` использовался тип `pthread_mutex_t`, а не семафор. Заметна ли разница в скорости работы программы?

11. Сравните быстродействие именованных семафоров (листинги 10.8 и 10.9) и размещаемых в памяти (листинг 10.11).

11.1. Введение

В главе 10 мы описывали различные виды семафоров, начав с:

- бинарного семафора, который может принимать только два значения: 0 и 1. По своим свойствам такой семафор аналогичен взаимному исключению (глава 7), причем значение 0 для семафора соответствует блокированию ресурса, а 1 — освобождению.

Далее мы перешли к более сложному виду семафоров:

- семафор-счетчик, значение которого лежит в диапазоне от 0 до некоторого ограничения, которое, согласно Posix, не должно быть меньше 32767. Они использовались для подсчета доступных ресурсов в задаче о производителях и потребителях, причем значение семафора соответствовало количеству доступных ресурсов.

Для обоих типов семафоров операция *wait* состояла в ожидании изменения значения семафора с нулевого на ненулевое и последующем уменьшении этого значения на 1. Операция *post* увеличивала значение семафора на 1, оповещая об этом все процессы, ожидающие изменения значения семафора.

Для семафоров System V определен еще один уровень сложности:

- набор семафоров-счетчиков — один или несколько семафоров, каждый из которых является счетчиком. На количество семафоров в наборе существует ограничение (обычно порядка 25 — раздел 11.7). Когда мы говорим о семафоре System V, мы подразумеваем именно набор семафоров-счетчиков, а когда говорим о семафоре Posix, подразумевается ровно один семафор-счетчик.

Для каждого набора семафоров ядро поддерживает следующую информационную структуру, определенную в файле `<sys/sem.h>`:

Структура `ipc_perm` была описана в разделе 3.3. Она содержит разрешения доступа для данного семафора.

Структура `sem` представляет собой внутреннюю структуру данных, используемую ядром для хранения набора значений семафора. Каждый элемент набора семафоров описывается так:

Обратите внимание, что `sem_base` представляет собой указатель на массив структур типа `sem` — по одному элементу массива на каждый семафор в наборе.

Помимо текущих значений всех семафоров набора в ядре хранятся еще три поля данных для каждого семафора: идентификатор процесса, изменившего значение семафора последним, количество процессов, ожидающих увеличения значения семафора, и количество процессов, ожидающих того, что значение семафора станет нулевым.

ПРИМЕЧАНИЕ

В стандарте Unix 98 данная структура не имеет имени. Приведенное выше имя (`sem`) взято из реализации System V.

Любой конкретный семафор в ядре мы можем воспринимать как структуру `semid_ds`, указывающую на массив структур `sem`. Если в наборе два элемента, мы получим картину, изображенную на рис. 11.1. На этом рисунке переменная `sem_nsems` имеет значение 2, а каждый из элементов набора идентифицируется индексом ([0] или [1]).



Рис. 11.1. Структуры данных ядра для набора семафоров из двух элементов

Функция `semget` создает набор семафоров или обеспечивает доступ к существующему.

Эта функция возвращает целое значение, называемое идентификатором семафора, которое затем используется при вызове функций `semop` и `semctl`.

Аргумент `nsems` задает количество семафоров в наборе. Если мы не создаем новый набор, а устанавливаем доступ к существующему, этот аргумент может быть нулевым. Количество семафоров в уже созданном наборе изменить нельзя.

Аргумент `oflag` представляет собой комбинацию констант `SEM_R` и `SEM_A` из табл. 3.3. Здесь `R` обозначает *Read* (чтение), а `A` — *Alter* (изменение). К этим константам можно логически прибавить `IPC_CREAT` или `IPC_CREAT | IPC_EXCL`, о чем мы уже говорили в связи с рис. 3.2.

При создании нового семафора инициализируются следующие поля структуры `semid_ds`:

- поля `uid` и `cuid` структуры `sem_perm` устанавливаются равными действующему идентификатору пользователя процесса, а поля `guid` и `cgid` устанавливаются равными действующему идентификатору группы процесса;
- биты разрешений чтения-записи аргумента `oflag` сохраняются в `sem_perm.mode`;
- поле `sem_otime` устанавливается в 0, а поле `sem_ctime` устанавливается равным текущему времени;
- значение `sem_nsems` устанавливается равным `nsems`;
- структуры `sem` для каждого из семафоров набора не инициализируются. Это происходит лишь при вызове `semctl` с командами `SETVAL` или `SETALL`.

Инициализация значения семафора

В комментариях к исходному коду в издании этой книги 1990 года неправильно утверждалось, что значения семафоров набора инициализируются нулем при вызове semget с созданием нового семафора. Хотя в некоторых системах это действительно происходит, гарантировать подобное поведение ядра нельзя. Более старые реализации System V вообще не инициализировали значения семафоров, оставляя их содержимое таким, каким оно было до выделения памяти.

В большинстве версий документации ничего не говорится о начальных значениях семафоров при создании нового набора. Руководство по написанию переносимых программ X/Open XPG3 (1989) и стандарт Unix 98 исправляют это упущение и открыто утверждают, что значения семафоров не инициализируются вызовом semget, а устанавливаются только при вызове semctl (вскоре мы опишем эту функцию) с командами SETVAL (установка значения одного из семафоров набора) и SETALL (установка значений всех семафоров набора).

Необходимость вызова двух функций для создания (semget) и инициализации (semctl) набора семафоров является неизправимым недостатком семафоров System V. Эту проблему можно решить частично, указывая флаги IPC_CREAT | IPC_EXCL при вызове semget, чтобы только один процесс, вызвавший semget первым, создавал семафор, и этот же процесс должен семафор инициализировать.

Другие процессы получают при вызове semget ошибку EEXIST, так что им приходится еще раз вызывать semget, уже не указывая флагов IPC_CREAT или IPC_EXCL.

Однако ситуация гонок все еще не устранена. Предположим, что два процесса пытаются создать и инициализировать набор семафоров с одним элементом приблизительно в один и тот же момент времени, причем оба они будут выполнять один и тот же фрагмент кода:

При этом может произойти вот что:

1. Первый процесс выполняет строки 1-3, а затем останавливается ядром.
2. Ядро запускает второй процесс, который выполняет строки 1, 2, 5, 6 и 9.

Хотя первый процесс, создавший семафор, и будет единственным процессом, который проинициализирует семафор, ядро может переключиться на другой процесс в промежутке между созданием и инициализацией семафора, и тогда второй процесс сможет обратиться к семафору (строка 9), который еще не был проинициализирован. Значение семафора после выполнения строки 9 для второго процесса будет не определено.

ПРИМЕЧАНИЕ

В семафорах Posix эта проблема исключается благодаря тому, что семафоры создаются и инициализируются единственным вызовом — sem_open. Более того, даже если указан флаг O_CREAT, семафор будет проинициализирован только в том случае, если он еще не существовал на момент вызова функции.

Будет ли обсуждавшаяся выше ситуация гонок создавать какие-то проблемы — зависит от приложения. В некоторых приложениях (например, задача производителей и потребителей в листинге 10.12) единственный процесс всегда создает и инициализирует семафор. В этом варианте ситуация гонок возникать не будет. В других приложениях (пример с блокировкой файлов в листинге 10.10) нет такого единственного процесса, который бы создавал и инициализировал семафор: первый процесс, открывающий семафор, должен создать его и проинициализировать, так что в этом случае ситуацию гонок следует исключать.

К счастью, существует способ исключить в данном случае ситуацию гонок. Стандарт гарантирует, что при создании набора семафоров поле sem_otime структуры semid_ds инициализируется нулем. (Руководства System V с давних пор говорят об этом, это утверждается и в стандартах XPG3 и Unix 98.) Это поле устанавливается равным текущему времени только при успешном вызове semop. Следовательно, второй процесс в приведенном выше примере должен просто вызвать semctl с командой IPC_STAT после второго вызова semget (строка 6). Затем этот процесс должен ожидать изменения значения sem_otime на ненулевое, после чего он может быть уверен в том, что семафор был успешно проинициализирован другим процессом. Это значит, что создавший семафор процесс должен проинициализировать его значение и успешно вызвать semop, прежде чем другие процессы смогут воспользоваться этим семафором. Мы используем этот метод в листингах 10.37 и 11.6.

11.3. Функция semop

После инициализации семафора вызовом `semget` с одним или несколькими семафорами набора можно выполнять некоторые действия с помощью функции `semop`:

Указатель `opsptr` указывает на массив структур вида

Количество элементов в массиве структур `sembuf`, на который указывает `opsptr`, задается аргументом `nops`. Каждый элемент этого массива определяет операцию с одним конкретным семафором набора. Номер семафора указывается в поле `sem_num` и принимает значение 0 для первого семафора, 1 для второго и т. д., до `nsems-1`, где `nsems` соответствует количеству семафоров в наборе (второй аргумент в вызове `semget` при создании семафора).

ПРИМЕЧАНИЕ

В структуре гарантированно содержатся только три указанных выше поля. Однако в ней могут быть и другие поля, причем порядок их может быть совершенно произвольным. Поэтому не следует статически инициализировать эту структуру кодом наподобие

Вместо этого следует инициализировать ее динамически, как в нижеследующем примере:

Весь массив операций, передаваемый функции `semop`, выполняется ядром как одна операция; атомарность при этом гарантируется. Ядро выполняет все указанные операции или ни одну из них. Пример на эту тему приведен в разделе 11.5.

Каждая операция задается значением `sem_op`, которое может быть отрицательным, нулевым или положительным. Сделаем несколько утверждений, которыми будем пользоваться при дальнейшем обсуждении:

- `semval` — текущее значение семафора (рис. 11.1);
- `semncnt` — количество потоков, ожидающих, пока значение семафора не станет больше текущего (рис. 11.1);
- `semzcnt` — количество потоков, ожидающих, пока значение семафора не станет нулевым (рис. 11.1);
- `semadj` — корректировочное значение данного семафора для вызвавшего процесса. Это значение обновляется, только если для данной операции указан флаг `SEM_UNDO` в поле `sem_flg` структуры `sembuf`. Эта переменная создается в ядре для каждого указавшего флаг `SEM_UNDO` процесса в отдельности; поле структуры с именем `semadj` не обязательно должно существовать;
- когда выполнение потока приостанавливается до завершения операции с семафором (мы увидим, что поток может ожидать либо обнуления семафора, либо получения семафором положительного значения), поток перехватывает сигнал и происходит возвращение из обработчика сигнала, функция `semop` возвращает ошибку `EINTR`. Используя терминологию, введенную в книге [24, с. 124], можно сказать, что функция `semop` представляет собой медленный системный вызов, который прерывается перехватываемыми сигналами;
- когда выполнение потока приостанавливается до завершения операции с семафором и этот семафор удаляется из системы другим потоком или процессом, функция `semop` возвращает ошибку `EIDRM` (`identifier removed` — идентификатор удален).

Опишем теперь работу функции `semop` в зависимости от трех возможных значений поля `sem_op`: отрицательного, нулевого и положительного.

1. Если значение `sem_op` положительно, оно добавляется к `semval`. Такое действие соответствует освобождению ресурсов, управляемых семафором. Если указан флаг `SEM_UNDO`, значение `sem_op` вычитается из значения `semadj` данного семафора.
2. Если значение `semop` равно нулю, вызвавший поток блокируется до тех пор, пока значение семафора (`semval`) не станет равным нулю. Если `semval` уже равно 0, происходит немедленное возвращение из функции.

Если `semval` не равно нулю, то ядро увеличивает значение поля `semzcnt` данного семафора и вызвавший поток блокируется до тех пор, пока значение `semval` не станет нулевым (после чего значение `semzcnt` будет уменьшено на 1). Как отмечалось ранее, поток будет приостановлен, только если не указан флаг `IPC_NOWAIT`. Если семафор будет удален в процессе ожидания либо будет перехвачен сигнал, произойдет преждевременный возврат из функции с возвращением кода ошибки.

3. Если значение `sem_op` отрицательно, вызвавший поток блокируется до тех пор, пока значение семафора не станет большим либо равным модулю `sem_op`. Это соответствует запрашиванию ресурсов.

Если значение `semval` больше либо равно модулю `sem_op`, модуль `sem_op` вычитается из `semval`. Если указан флаг `SEM_UNDO`, модуль `sem_op` добавляется к значению поля `semadj` данного семафора.

Если значение `semval` меньше модуля `sem_op`, значение поля `semncnt` данного семафора увеличивается, а вызвавший поток блокируется до тех пор, пока `semval` не станет больше либо равно модулю `semop`. Когда это произойдет, поток будет разблокирован, а модуль `sem_op` будет отнят от `semval` и из значения `semncnt` будет вычтена единица. Если указан флаг `SEM_UNDO`, модуль `sem_op` добавляется к значению поля `semadj` данного семафора. Как отмечалось ранее, поток не будет приостановлен, если указан флаг `IPC_NOWAIT`. Ожидание завершается преждевременно, если перехватываемый сигнал вызывает прерывание либо семафор удаляется другим потоком.

ПРИМЕЧАНИЕ

Если сравнить этот набор операций с теми, которые разрешены для семафоров Posix, мы увидим, что для последних определены только команды `-1` (`sem_wait`) и `+1` (`sem_post`). Для семафоров System V значение семафора может изменяться с шагом, отличным от 1, и кроме того, поток может ожидать, чтобы значение семафора стало нулевым. Эти операции являются более общими, что вместе с возможностью включения нескольких семафоров в набор делает семафоры System V более сложными, чем одиночные семафоры Posix.

11.4. Функция semctl

Функция `semctl` предназначена для выполнения разного рода вспомогательных управляющих операций с семафорами.

Первый аргумент (`semid`) представляет собой идентификатор семафора, а `semnum` указывает элемент набора семафоров (0, 1 и т. д. до `nsems` - 1). Значение `semnum` используется только командами `GETVAL`, `SETVAL`, `GETNCNT`, `GETZCNT` и `GETPID`.

Четвертый аргумент является дополнительным — он добавляется в зависимости от команды `cmd` (см. комментарии в описании объединения). Объявляется это объединение следующим образом:

Это объединение отсутствует в системных заголовочных файлах и должно декларироваться приложением (мы определяем его в заголовочном файле `iprsrc.h`, листинг B.1). Оно передается по значению, а не по ссылке, то есть аргументом является собственно значение объединения, а не указатель на него.

ПРИМЕЧАНИЕ

К сожалению, в некоторых системах (FreeBSD и Linux) это объединение определено в заголовочном файле `<sys/sem.h>`, что затрудняет написание переносимых программ. Хотя в объявлении этого объединения в системном заголовочном файле и есть некоторый смысл, стандарт Unix 98 требует, чтобы оно каждый раз явно объявлялось приложением.

Ниже приведен список поддерживаемых значений аргумента `cmd`. В случае успешного завершения функция возвращает 0, а в случае ошибки -1, если в описании команды не сказано что-либо другое.

- `GETVAL` — возвращает текущее значение `semval`. Поскольку значение семафора отрицательным быть не может (`semval` объявляется как `unsigned short` — беззнаковое короткое целое), в случае успешного возврата значение всегда будет неотрицательным.
- `SETVAL` — установка значения `semval` равным `arg.val`. В случае успешного выполнения корректировочное значение этого семафора (`semadj`) устанавливается равным нулю для всех процессов.
- `GETPID` — функция возвращает текущее значение поля `semid`.
- `GETNCNT` — функция возвращает текущее значение поля `semncnt`.
- `GETZCNT` — функция возвращает текущее значение поля `semzcnt`.
- `GETALL` — возвращаются значения `semval` для всех элементов набора. Значения возвращаются через указатель `arg.agray`, а сама функция при этом возвращает 0. Обратите внимание, что вызывающий процесс должен самостоятельно выделить массив беззнаковых коротких целых достаточного объема для хранения всех значений семафоров набора, а затем сделать так, чтобы `arg.agray` указывал на этот массив.
- `SETALL` — установка значений `semval` для всех элементов набора. Значения задаются через указатель `arg.agray`.
- `IPC_RMID` — удаление набора семафоров, задаваемого через идентификатор `semid`.
- `IPC_SET` — установка трех полей структуры `semid_ds` равными соответствующим полям структуры `arg.buf`: `sem_perm.uid`, `sem_perm.gid` и `sem_perm.mode`. Поле `sem_ctime` структуры `semid_ds` устанавливается равным текущему времени.
- `IPC_STAT` — возвращение вызвавшему процессу через аргумент `arg.buf` текущего значения полей структуры `semid_ds` для данного набора семафоров. Обратите внимание, что вызывающий процесс должен сначала выделить место под структуру `semid_ds` и установить на нее указатель `arg.buf`.

Поскольку семафоры System V обладают живучестью ядра, мы можем продемонстрировать работу с ними, написав несколько небольших программ, которые будут выполнять с семафорами различные действия. В промежутках между выполнением отдельных программ значения семафоров будут храниться в ядре.

Программа semcreate

Первая программа, текст которой приведен в листинге 11.1, просто создает набор семафоров System V. Параметр командной строки -e соответствует флагу IPC_EXCL при вызове semget, а последним аргументом командной строки является количество семафоров в создаваемом наборе.

Программа semrmid

Следующая программа, текст которой приведен в листинге 11.2, удаляет набор семафоров из системы. Для этого используется вызов semctl с командой (аргументом cmd) IPC_RMID.

Программа semsetvalues

Программа semsetvalues (листинг 11.3) устанавливает значения всех семафоров набора.

11-15 После получения идентификатора семафора с помощью semget мы вызываем semctl с командой IPC_STAT, чтобы получить значения полей структуры semid_ds для данного семафора. Поле sem_nsems содержит нужную нам информацию о количестве семафоров в наборе.

19-24 Мы выделяем память под массив беззнаковых коротких целых, по одному элементу на каждый семафор набора, затем копируем полученные из командной строки значения в этот массив. Вызов semctl с командой SETALL позволяет установить все значения семафоров набора одновременно.

Программа semgetvalues

В листинге 11.4 приведен текст программы `semgetvalues`, которая получает и выводит значения всех семафоров набора.

11-15 После получения идентификатора семафора с помощью `semget` мы вызываем `semctl` с командой `IPC_STAT` для получения значений полей структуры `semid_ds` данного семафора. Поле `sem_nsems` содержит нужную нам информацию о количестве семафоров в наборе.

16-22 Мы выделяем память под массив беззнаковых коротких целых, по одному элементу на каждый семафор набора. Вызов `semctl` с командой `GETALL` позволяет получить все значения семафоров набора одновременно. Каждое значение выводится.

Программа semops

В листинге 11.5 приведен текст программы `semops`, позволяющей выполнять последовательность действий над набором семафоров.

7-19 Параметр `-n` соответствует установленному флагу `IPC_NOWAIT` для каждой операции, а параметр `-i` аналогичным образом позволяет указать для каждой операции флаг `SEM_UNDO`. Обратите внимание, что функция `semop` позволяет указывать свой набор флагов для каждого элемента структуры `sembuf` (то есть для каждой из операций в отдельности), но для простоты мы в нашей программе задаем одинаковые флаги для всех операций.

20-29 После открытия семафора вызовом `semget` мы выделяем память под массив структур `sembuf`, по одному элементу на каждую операцию из командной строки. В отличие от предыдущих двух программ эта позволяет пользователю задать меньше команд, чем имеется семафоров в наборе.

30 Вызов `semop` выполняет последовательность операций над семафорами набора.

Примеры

Теперь мы продемонстрируем работу пяти приведенных выше программ и исследуем некоторые свойства семафоров System V:

Сначала мы создали файл с именем /tmp/rich, который использовался при вызове ftok для вычисления идентификатора набора семафоров. Программа semcreate создает набор с тремя элементами. Программа semsetvalues устанавливает значения этих элементов (1, 2 и 3), а semgetvalues выводит их значения.

Теперь продемонстрируем атомарность выполнения последовательности операций над набором:

В командной строке мы указываем параметр, отключающий блокировку (-n), и три операции, каждая из которых уменьшает одно из значений набора семафоров. Первая операция завершается успешно (мы можем вычесть 1 из значения первого элемента набора, потому что до вычитания оно равно 1), вторая операция также проходит (вычитаем 2 из значения второго семафора, равного 2), но третья операция выполнена быть не может (мы не можем вычесть 4 из значения третьего семафора, потому что оно равно 3). Поскольку последняя операция последовательности не может быть выполнена и поскольку мы отключили режим блокирования процесса, функция возвращает ошибку EAGAIN. Если бы мы не указали флаг отключения блокировки, выполнение процесса было бы приостановлено до тех пор, пока операция вычитания не стала бы возможной. После этого мы проверяем, чтобы ни одно из значений семафоров набора не изменилось. Хотя первые две операции и могли бы быть выполнены, ни одна из трех на самом деле произведена не была, поскольку последняя операция была некорректной. Атомарность semop и означает, что выполняются либо все операции, либо ни одна из них.

Теперь продемонстрируем работу флага SEM_UNDO:

Сначала мы заново устанавливаем значения семафоров в наборе равными 1, 2 и 3 с помощью программы semsetvalues, а затем запускаем программу semops с операциями -1, -2, -3. При этом все три значения семафоров становятся нулевыми, но, так как мы указали параметр -u при вызове semops, для всех трех операций устанавливается флаг SEM_UNDO. При этом значения semadj для элементов набора семафоров становятся равными 1, 2 и 3 соответственно. После завершения программы semops эти значения добавляются к значениям семафоров, в результате чего их значения становятся равными 1, 2 и 3, как будто мы и не запускали программу. В этом мы убеждаемся, запустив semgetvalues. Затем мы снова запускаем semops, но уже без параметра -u, и убеждаемся, что при этом значения семафоров становятся нулевыми и остаются таковыми даже после выхода из программы.

11.6. Блокирование файлов

С помощью семафоров System V можно реализовать еще одну версию функций `my_lock` и `my_unlock` из листинга 10.10. Новый вариант приведен в листинге 11.6.

13-17 Нам нужно гарантировать, что только один процесс проинициализирует семафор, поэтому при вызове `semget` мы указываем флаги `IPC_CREAT | IPC_EXCL`. Если этот вызов оказывается успешным, процесс вызывает `semctl` для инициализации семафора значением 1. Если мы запустим несколько процессов одновременно и все они вызовут функцию `my_lock`, только один из них создаст семафор (предполагается, что он еще не существует) и проинициализирует его.

18-20 Если первый вызов `semget` возвращает ошибку `EEXIST`, процесс вызывает `semget` еще раз, но уже без флагов `IPC_CREAT` и `IPC_EXCL`.

21-28 В этой программе возникает такая же ситуация гонок, как и обсуждавшаяся в разделе 11.2, когда мы говорили об инициализации семафоров System V вообще. Для исключения такой ситуации все процессы, которые обнаруживают, что семафор уже создан, вызывают `semctl` с командой `IPC_STAT`, проверяя значение `sem_otime` данного семафора. Когда это значение становится ненулевым, мы можем быть уверены, что создавший семафор процесс проинициализировал его и вызвал `semop` (этот вызов находится в конце функции) успешно. Если значение этого поля оказывается нулевым (что должно происходить крайне редко), мы приостанавливаем выполнение процесса на одну секунду вызовом `sleep`, а затем повторяем попытку. Число попыток мы ограничиваем, чтобы процесс не «заснул» навсегда.

33-38 Как отмечалось ранее, конкретный порядок полей структуры `sembuf` зависит от реализации, поэтому статически инициализировать ее нельзя. Вместо этого мы выделяем место под две такие структуры и присваиваем значения их полям во время выполнения программы, когда процесс вызывает `my_lock` в первый раз. При этом мы указываем флаг `SEM_UNDO`, чтобы ядро сняло блокировку, если процесс завершит свою работу, не сняв ее самостоятельно (см. упражнение 10.3).

Создание семафора при первой необходимости реализовать довольно просто (все процессы пытаются создать семафор, игнорируя ошибку, если он уже существует), но удаление семафора после завершения работы всех процессов организовать гораздо сложнее. В случае демона печати, использующего файл с последовательным номером для упорядочения заданий печати, удалять семафор нет необходимости. Но в других приложениях может возникнуть необходимость удалить семафор при удалении соответствующего файла. В этом случае лучше пользоваться блокировкой записи, чем семафором.

На семафоры System V накладываются определенные системные ограничения, так же, как и на очереди сообщений. Большинство этих ограничений были связаны с особенностями реализации System V (раздел 3.8). Они показаны в табл. 11.1. Первая колонка содержит традиционное для System V имя переменной ядра, в которой хранится соответствующее ограничение.

Таблица 11.1. Типичные значения ограничений для семафоров System V

В Digital Unix 4.0B никакого ограничения на semmnu не существует.

Пример

Программа в листинге 11.7 позволяет определить ограничения, приведенные в табл. 11.1.

11.8. Резюме

У семафоров System V имеются следующие отличия от семафоров Posix:

1. Семафоры System V представляют собой набор значений. Последовательность операций над набором семафоров либо выполняется целиком, либо не выполняется вовсе.
2. К любому элементу набора семафоров могут быть применены три операции: проверка на нулевое значение, добавление некоторого значения к текущему и вычитание некоторого значения из текущего (в предположении, что значение остается неотрицательным). Для семафоров Posix определены только операции увеличения и уменьшения значения семафора на 1 (в предположении, что значение остается неотрицательным).
3. Создание семафора System V имеет некоторую особенность, заключающуюся в необходимости выполнения двух вызовов для создания и инициализации семафора, что может привести к ситуации гонок.
4. Семафоры System V предоставляют возможность отмены операции с ними (*undo*) после завершения работы процесса.

Упражнения

1. Листинг 6.6 представлял собой измененный вариант листинга 6.4, в котором программа принимала идентификатор очереди вместо полного имени файла. Мы продемонстрировали, что для получения доступа к очереди System V достаточно знать только ее идентификатор (предполагается наличие достаточных разрешений). Проделайте аналогичные изменения с программой в листинге 11.5 и посмотрите, верно ли вышесказанное для семафоров System V.

2. Что произойдет с программой в листинге 11.6, если файл LOCK_PATH не будет существовать?

12.1. Введение

Разделяемая память является наиболее быстрым средством межпроцессного взаимодействия. После отображения области памяти в адресное пространство процессов, совместно ее использующих, для передачи данных между процессами больше не требуется участие ядра. Обычно, однако, требуется некоторая форма синхронизации процессов, помещающих данные в разделяемую память и считающих ее оттуда. В части 3 мы обсуждали различные средства синхронизации: взаимные исключения, условные переменные, блокировки чтения-записи, блокировки записей и семафоры.

ПРИМЕЧАНИЕ

Говоря «не требуется участие ядра», мы подразумеваем, что процессы не делают системных вызовов для передачи данных. Очевидно, что все равно именно ядро обеспечивает отображение памяти, позволяющее процессам совместно ею пользоваться, и затем обслуживает эту память (обрабатывает сбои страниц и т. п.).

Рассмотрим по шагам работу программы копирования файла типа клиент-сервер, которую мы использовали в качестве примера для иллюстрации различных способов передачи сообщений (рис. 4.1).

- Сервер считывает данные из входного файла. Данные из файлачитываются ядром в свою память, а затем копируются из ядра в память процесса.
- Сервер составляет сообщение из этих данных и отправляет его, используя именованный или неименованный канал или очередь сообщений. Эти формы IPC обычно требуют копирования данных из процесса в ядро.

ПРИМЕЧАНИЕ

Мы говорим «обычно», поскольку очереди сообщений Posix могут быть реализованы через отображение файла в память (функцию mmap мы опишем в этой главе), как мы показали в разделе 5.8 и в решении упражнения 12.2. На рис. 12.1 мы предполагаем, что очереди сообщений Posix реализованы в ядре, что также возможно. Но именованные и неименованные каналы и очереди сообщений System V требуют копирования данных из процесса в ядро вызовом write или msgsnd или копирования данных из ядра процессу вызовом read или msgrcv.

- Клиент считывает данные из канала IPC, что обычно требует их копирования из ядра в пространство процесса.
- Наконец, данные копируются из буфера клиента (второй аргумент вызова write) в выходной файл.

Таким образом, для копирования файла обычно требуются четыре операции копирования данных. К тому же эти операции копирования осуществляются между процессами и ядром, что часто является дорогостоящей операцией (более дорогостоящей, чем копирование данных внутри ядра или внутри одного процесса). На рис. 12.1 изображено перемещение данных между клиентом и сервером через ядро.



Рис. 12.1. Передача содержимого файла от сервера к клиенту

Недостатком этих форм IPC — именованных и неименованных каналов — является то, что для передачи между процессами информация должна пройти через ядро.

Разделяемая память дает возможность обойти этот недостаток, поскольку ее использование позволяет двум процессам обмениваться данными через общий участок памяти. Процессы, разумеется, должны синхронизировать и координировать свои действия. Одновременное использование участка памяти во многом аналогично совместному доступу к файлу, например к файлу с последовательным номером, который фигурировал во всех примерах на блокировку доступа к файлам. Для синхронизации такого рода может применяться любой из методов, описанных в третьей части книги.

Теперь информация передается между клиентом и сервером в такой последовательности:

- сервер получает доступ к объекту разделяемой памяти, используя для синхронизации семафор (например);
- сервер считывает данные из файла в разделяемую память. Второй аргумент вызова read (адрес

буфера) указывает на объект разделяемой памяти;

- после завершения операции считывания клиент уведомляется сервером с помощью семафора;
- клиент записывает данные из объекта разделяемой памяти в выходной файл.



Рис. 12.2. Копирование файла через разделяемую память

Этот сценарий иллюстрирует рис. 12.2.

Из этого рисунка видно, что копирование данных происходит всего лишь дважды: из входного файла в разделяемую память и из разделяемой памяти в выходной файл. Мы нарисовали два прямоугольника штриховыми линиями; они подчеркивают, что разделяемая память принадлежит как адресному пространству клиента, так и адресному пространству сервера.

Концепции, связанные с использованием разделяемой памяти через интерфейсы Posix и System V, похожи. Первый интерфейс описан в главе 13, а второй — в главе 14.

В этой главе мы возвращаемся к примеру с увеличением последовательного номера, который впервые появился в главе 9. Теперь мы будем хранить последовательный номер в сегменте разделяемой памяти, а не в файле.

Сначала мы подчеркнем, что память разделяется между родительским и дочерним процессами при вызове `fork`. В программе из листинга 12.1 родительский и дочерний процессы по очереди увеличивают глобальный целочисленный счетчик `count`.

12-14 Мы создаем и инициализируем семафор, защищающий переменную, которую мы считаем глобальной (`count`). Поскольку предположение о ее глобальности ложно, этот семафор на самом деле не нужен. Обратите внимание, что мы удаляем семафор из системы вызовом `sem_unlink`, но хотя файл с соответствующим полным именем при этом и удаляется, на открытый в данный момент семафор эта команда не действует. Этот вызов мы делаем для того, чтобы файл был удален даже при досрочном завершении программы.

15 Мы отключаем буферизацию стандартного потока вывода, поскольку запись в него будет производиться и родительским, и дочерним процессами. Это предотвращает смешивание вывода из двух процессов.

16-29 Родительский и дочерний процессы увеличивают глобальный счетчик в цикле заданное число раз, выполняя операции только при установленном семафоре.

Если мы запустим эту программу на выполнение и посмотрим на результат, обращая внимание только на те строки, где система переключается между родительским и дочерним процессами, мы увидим вот что:

Как видно, каждый из процессов использует собственную копию глобального счетчика `count`. Каждый начинает со значения 0 и при прохождении цикла увеличивает значение своей копии счетчика. На рис. 12.3 изображен родительский процесс перед вызовом `fork`.



Рис. 12.3. Родительский процесс перед вызовом `fork`

При вызове `fork` дочерний процесс запускается с собственной копией данных родительского процесса. На рис. 12.4 изображены оба процесса после возвращения из `fork`.



Рис. 12.4. Родительский и дочерний процессы после возвращения из `fork`

Мы видим, что родительский и дочерний процессы используют отдельные копии счетчика `count`.

Функция `mmap` отображает в адресное пространство процесса файл или объект разделяемой памяти Posix. Мы используем эту функцию в следующих ситуациях:

1. С обычными файлами для обеспечения ввода-вывода через отображение в память (раздел 12.3).
2. Со специальными файлами для обеспечения неименованного отображения памяти (разделы 12.4 и 12.5).
3. С `shm_open` для создания участка разделяемой неродственными процессами памяти Posix.

Аргумент `addr` может указывать начальный адрес участка памяти процесса, в который следует отобразить содержимое дескриптора `fd`. Обычно ему присваивается значение нулевого указателя, что говорит ядру о необходимости выбрать начальный адрес самостоятельно. В любом случае функция возвращает начальный адрес сегмента памяти, выделенной для отображения.

Аргумент `len` задает длину отображаемого участка в байтах; участок может начинаться не с начала файла (или другого объекта), а с некоторого места, задаваемого аргументом `offset`. Обычно `offset = 0`. На рис. 12.5 изображена схема отображения объекта в память.



Рис. 12.5. Пример отображения файла в память

Захист участка памяти с отображенном об'єктом обслуговується з допомогою аргумента `prot` і констант, приведених в табл. 12.1. Обичне значення цього аргумента — `PROT_READ | PROT_WRITE`, що обслуговує доступ на читання і запис.

Таблица 12.1. Аргумент `prot` для вызова `mmap`

Таблица 12.2. Аргумент `flag` для вызова `mmap`

Аргумент `flags` може приймати значення з табл. 12.2. Можна вказати тільки один із флагів — `MAP_SHARED` або `MAP_PRIVATE`, додавши до нього `MAP_FIXED`. Якщо вказан флаг `MAP_PRIVATE`, всі зміни будуть проводитися тільки з об'єкта в адресному просторі процеса; іншим процесам вони недоступні. Якщо ж вказан флаг `MAP_SHARED`, зміни, внесені в отображені дані, будуть видні всім процесам, що спільно використовують об'єкт.

Для обслуговування переносимості програм флаг `MAP_FIXED` вказувати не слід. Якщо він не вказан, але аргумент `addr` представляє собою ненулевий указатель, інтерпретація цього аргумента залежить від реалізації. Ненулеве значення `addr` зазвичай трактується як указатель на потрібну область пам'яті, в яку треба зробити отображення. В переносимій програмі значення `addr` повинно бути нулевим і флаг `MAP_FIXED` не має бути вказан.

Одним із способів здобуття спільного використання пам'яті батьківським і дочернім процесами є виклик `mmap` з флагом `MAP_SHARED` перед викликом `fork`. Стандарт Posix.1 гарантує в цьому випадку, що всі отображені пам'яті, створені батьківським процесом, будуть унаслідувані дочернім. Більше того, зміни в контенті об'єкта, внесені батьківським процесом, будуть видні дочерньому, і навпаки. Цю схему ми вскорі продемонструємо в дії.

Для відключення отображення об'єкта в адресне просторі процеса використовується виклик `munmap`:

Аргумент `addr` повинен містити адрес, возвращений `mmap`, а `len` — довжину області отображення. Після виклику `munmap` будь-які спроби обратитися до цієї області пам'яті приведуть до надання процесу сигналу `SIGSEGV` (загадується, що ця область пам'яті не буде знову отображена викликом `mmap`).

Якщо область була отображена з флагом `MAP_PRIVATE`, всі зміни, внесені за час роботи процеса, будуть скидані.

В зображеній на рис. 12.5 схемі ядро забезпечує синхронізацію контенту об'єкта, отображеного в пам'яті, з самим пам'ятю при допомозі алгоритма роботи з виртуальною пам'ятю (якщо сегмент був отображен з флагом `MAP_SHARED`). Якщо ми змінюємо контент ячейки

памяти, в которую отображен файл, через некоторое время содержимое файла будет соответствующим образом изменено ядром. Однако в некоторых случаях нам нужно, чтобы содержимое файла всегда было в соответствии с содержимым памяти. Тогда для осуществления моментальной синхронизации мы вызываем `msync`:

Аргумент *flags* представляет собой комбинацию констант из табл. 12.3.

Таблица 12.3. Значения аргумента *flags* для функции `msync`

Из двух констант `MS_ASYNC` и `MS_SYNC` указать нужно одну и только одну. Отличие между ними в том, что возврат из функции при указании флага `MS_ASYNC` происходит сразу же, как только данные для записи будут помещены в очередь ядром, а при указании флага `MS_SYNC` возврат происходит только после завершения операций записи. Если указан и флаг `MS_INVALIDATE`, все копии файла, содержимое которых не совпадает с его текущим содержимым, считаются устаревшими. Последующие обращения к этим копиям приведут к считыванию данных из файла.

Почему вообще используется отображение в память?

До сих пор мы всегда говорили об отображении в память содержимого файла, который сначала открывается вызовом open, а затем отображается вызовом mmap. Удобство состоит в том, что все операции ввода-вывода осуществляются ядром и скрыты от программиста, а он просто пишет код,читывающий и записывающий данные в некоторую область памяти. Ему не приходится вызывать read, write или lseek. Часто это заметно упрощает код.

ПРИМЕЧАНИЕ

Вспомните нашу реализацию очередей сообщений Posix с использованием mmap, где значения сохранялись в структуре msg_hdr и считывались из нее же (листинги 5.26 и 5.28).

Следует, однако, иметь в виду, что не все файлы могут быть отображены в память. Попытка отобразить дескриптор, указывающий на терминал или сокет, приведет к возвращению ошибки при вызове mmap. К дескрипторам этих типов доступ осуществляется только с помощью read и write (и аналогичных вызовов).

Другой целью использования mmap может являться разделение памяти между неродственными процессами. В этом случае содержимое файла становится начальным содержимым разделяемой памяти и любые изменения, вносимые в нее процессами, копируются обратно в файл (что дает этому виду IPC живучесть файловой системы). Предполагается, что при вызове mmap указывается флаг MAP_SHARED, необходимый для разделения памяти между процессами.

ПРИМЕЧАНИЕ

Детали реализации mmap и связь этого вызова с механизмами реализации виртуальной памяти описаны в [14] для 4.4BSD и [6] для SVR4.

12.3. Увеличение счетчика в отображаемом в память файле

Изменим программу в листинге 12.1 (которая не работала) таким образом, чтобы родительский и дочерний процессы совместно использовали область памяти, в которой хранится счетчик. Для этого используем отображение файла в память вызовами `open` и `mmap`. В листинге 12.2 приведен текст новой программы.

11-14 Из командной строки теперь считывается еще один аргумент, задающий полное имя файла, который будет отображен в память. Мы открываем файл для чтения и записи, причем если файл не существует, он будет создан, а затем инициализируем файл нулем.

15-16 Вызов `mmap` позволяет отобразить открытый файл в адресное пространство процесса. Первый аргумент является нулевым указателем, при этом система сама выбирает адрес начала отображаемого сегмента. Длина файла совпадает с размером целого числа. Устанавливается доступ на чтение и запись. Четвертый аргумент имеет значение `MAP_SHARED`, что позволяет процессам «видеть» изменения, вносимые друг другом. Функция возвращает адрес начала участка разделяемой памяти, мы сохраняем его в переменной `ptr`.

20-34 Мы отключаем буферизацию стандартного потока вывода и вызываем `fork`. И родительский, и дочерний процессы по очереди увеличивают значение целого, на которое указывает `ptr`.

Отображенные в память файлы обрабатываются при вызове `fork` специфическим образом в том смысле, что созданные родительским процессом отображения наследуются дочерним процессом. Следовательно, открыв файл и вызвав `mmap` с флагом `MAP_SHARED`, мы получили область памяти, совместно используемую родительским и дочерним процессами. Более того, поскольку эта общая область на самом деле представляет собой отображенный файл, все изменения, вносимые в нее (область памяти, на которую указывает `ptr`, — размером `sizeof(int)`), также действуют и на содержимое реального файла (имя которого было указано в командной строке).

Запустив эту программу на выполнение, мы увидим, что память, на которую указывает `ptr`, действительно используется совместно родительским и дочерним процессами. Приведем значения счетчика перед переключением процессов:

Поскольку использовалось отображение файла в память, мы можем взглянуть на его содержимое с помощью программы `od` и увидеть, что окончательное значение счетчика (20000) действительно было сохранено в файле.

На рис. 12.6 изображена схема, отличающаяся от рис. 12.4. Здесь используется разделяемая память и показано, что семафор также используется совместно. Семафор мы изобразили размещенным в ядре, но для семафоров Posix это не обязательно. В зависимости от реализации семафор может обладать различной живучестью, но она должна быть по крайней мере не меньше живучести ядра. Семафор может быть реализован также через отображение файла в память, что мы продемонстрировали в разделе 10.15.



Рис. 12.6. Родительский и дочерний процессы используют разделяемую память и общий семафор

Мы показали, что у родительского и дочернего процессов имеются собственные копии указателя `ptr`, но обе копии указывают на одну и ту же область памяти — счетчик, увеличиваемый обоими процессами.

Изменим программу в листинге 12.2 так, чтобы использовались семафоры Posix, размещаемые в памяти (вместо именованных). Разместим такой семафор в разделяемой области памяти. Новая программа приведена в листинге 12.3.

2-5 Помещаем целочисленный счетчик и семафор, защищающий его, в одну структуру. Эта структура будет храниться в разделяемой памяти.

14-19 Создается файл для отображения в память, который инициализируется структурой с нулевым значением обоих полей. При этом инициализируется только счетчик, поскольку семафор будет инициализирован позднее вызовом `sem_init`. Тем не менее проще инициализировать всю структуру нулями, чем только одно целочисленное поле.

20-21 Используем семафор, размещаемый в памяти, вместо именованного. Для его инициализации единицей вызываем `sem_init`. Второй аргумент должен быть ненулевым, чтобы семафор мог совместно использоваться несколькими процессами.

На рис. 12.7 изображена модификация рис. 12.6, где семафор переместился из ядра в разделяемую память.



Рис. 12.7. И семафор, и счетчик теперь хранятся в разделяемой памяти

12.4. Неименованное отображение в память в 4.4BSD

Наши примеры из листингов 12.2 и 12.3 работают отлично, но нам приходится создавать файл в файловой системе (аргумент командной строки), вызывать open, записывать нули в файл вызовом write (чтобы проинициализировать его). Если mmap используется для передачи области разделяемой памяти через fork, мы можем упростить эту схему, используя свойства реализации.

1. В версии 4.4BSD предоставляется возможность неименованного отображения в память. При этом полностью пропадает необходимость создавать или открывать файлы. Вместо этого указываются флаги MAP_SHARED | MAP_ANON и дескриптор fd = -1. Сдвиг, задаваемый аргументом *offset*, игнорируется. Память автоматически инициализируется нулями. Пример использования приведен в листинге 12.4.

2. В версии SVR4 имеется файл /dev/zero, который мы открываем и дескриптор которого указываем при вызове mmap. Это устройство возвращает нули при попытке считывания, а весь направляемый на него вывод сбрасывается. Пример использования приведен в листинге 12.5. (Во многих реализациях, произошедших от BSD, также поддерживается устройство /dev/zero, например в SunOS 4.1.x и BSD/OS 3.1.)

В листинге 12.4 приведена часть листинга 12.2, которая изменяется при переходе к использованию неименованного отображения в память в 4.4BSD.

6-11 Автоматические переменные fd и zero больше не используются, как и аргумент командной строки, задававший имя создаваемого файла.

12-14 Файл больше не нужно открывать. Вместо этого указывается флаг MAP_ANON при вызове mmap, а пятый аргумент этой функции (дескриптор) принимает значение -1.

12.5. Отображение в память в SVR4 с помощью /dev/zero

В листинге 12.5 приведена часть новой версии программы, претерпевшая изменения по сравнению с листингом 12.2 при переходе к использованию отображения с помощью `/dev/zero`.

6-11 Автоматическая переменная `zero` больше не используется, как и аргумент командной строки, задававший имя создаваемого файла.

12-15 Мы открываем файл `/dev/zero` и передаем его дескриптор функции `mmap`. Область памяти будет гарантированно проинициализирована нулями.

12.6. Обращение к объектам, отображенными в память

Когда в память отображается обычный файл, размер полученной области (второй аргумент вызова mmap), как правило, совпадает с размером файла. Например, в листинге 12.3 размер файла устанавливается равным размеру структуры shared вызовом write и это же значение размера области используется при отображении его в память. Однако эти два параметра — размер файла и размер области памяти, в которую он отображен, — могут и отличаться.

Для детального изучения особенностей функции mmap воспользуемся программой в листинге 12.6.

8-11 Аргументы командной строки задают полное имя создаваемого и отображаемого в память файла, его размер и размер области памяти.

12-15 Если файл не существует, он будет создан. Если он существует, его длина будет установлена равной нулю. Затем размер файла устанавливается равным указанному размеру путем вызова lseek для установки текущей позиции, равной требуемому размеру минус 1 и записи 1 байта.

16-17 Файл отображается в память, причем размер области задается последним аргументом командной строки. Затем дескриптор файла закрывается.

18-19 Размер страницы памяти получается вызовом sysconf и выводится на экран.

20-26 Считываются и выводятся данные из области памяти, в которую отображен файл. Считываются первый и последний байты каждой страницы этой области памяти. Все значения должны быть нулевыми. Затем первый и последний байты каждой страницы устанавливаются в 1. Одно из обращений к памяти может привести к отправке сигнала процессу, что приведет к его завершению. После завершения цикла for мы добавляем еще одно обращение к следующей странице памяти, что должно заведомо привести к ошибке и завершению программы (если ошибка не возникла раньше).

Рассмотрим первую ситуацию: размер файла совпадает с размером области памяти, но эта величина не кратна размеру страницы памяти в данной реализации:

Размер страницы памяти составляет 4096 байт, и мы смогли обратиться ко всему содержимому второй страницы (индексы 4096-8191), но обращение к третьей странице (8192) приводит к отправке сигнала SIGSEGV, о чем интерпретатор оповещает сообщением Segmentation Fault. Хотя мы и установили значение ptr[8191] = 1, оно не было записано в файл и его размер остался равным 5000 байт. Ядро позволяет считывать и записывать данные в ту часть последней страницы, которая не относится к отображеному файлу (поскольку защита памяти осуществляется ядром постранично), но изменения в этой области памяти не будут скопированы в файл. А вот относящиеся к файлу изменения (индексы 0, 4095 и 4096) были скопированы в него, в чем мы убедились, воспользовавшись программой od (параметр -b при вызове последней указывает на необходимость выводить значения байтов в восьмеричном формате, а параметр -Ad позволяет выводить адреса в десятичном формате). На рис. 12.8 изображена схема памяти для данного примера.



Рис. 12.8. Размер отображаемого файла совпадает с размером области памяти

Запустив этот пример в Digital Unix 4.0B, получим тот же результат, но размер страницы памяти в этой системе равняется 8192 байт:

Мы все так же можем обратиться к памяти за пределами отображенного файла, но не выходя за границы страницы памяти (индексы с 5000 по 8191). Обращение к ptr[8192] приводит к отправке SIGSEGV, на что мы и рассчитывали.

Вторая ситуация: размер области памяти (15000 байт) превышает размер файла (5000 байт):



Рис. 12.9. Размер области памяти больше размера отображаемого файла

Полученный результат аналогичен результату предыдущего примера, в котором размер файла равнялся размеру области отображения (5000 байт). Однако в данном примере генерируется сигнал

SIGBUS (о чем интерпретатор оповещает сообщением Bus Error), тогда как в предыдущем примере отправлялся сигнал SIGSEGV. Отличие в том, что SIGBUS означает выход за границы отображеного файла внутри области отображения, а SIGSEGV — выход за границы области. Этим примером мы показали, что ядро хранит информацию о размере отображеного объекта, даже несмотря на то, что его дескриптор закрыт. Ядро позволяет указать при вызове mmap размер области памяти, больший размера файла, но не позволяет обратиться к адресам в этой области (кроме остатка последней страницы, в которой еще имеется содержимое собственно файла — индексы с 5000 по 8191 в данном случае). На рис. 12.9 приведена иллюстрация к этому примеру.

Следующая программа приведена в листинге 12.7. Ею мы иллюстрируем типичные методы работы с увеличивающимися в размерах файлами: при отображении в память заказывается большой размер области, текущий размер файла учитывается при всех операциях (чтобы не выйти за его пределы в памяти), а затем он просто увеличивается, по мере того как в файл записываются данные.

9-11 Мы создаем файл, если он еще не существует, или урезаем его до нулевой длины, если он существует. Затем файл отображается в область объемом 32 768 байт, хотя его текущий размер равен нулю.

12-16 Мы увеличиваем размер файла на 4096 байт за один вызов ftruncate (раздел 13.3) и считываем из него последний байт в каждом проходе цикла.

Запустив эту программу, мы убедимся в возможности обращаться к новым данным по мере роста файла:

Этот пример показывает, что ядро всегда следит за размером отображаемого в память объекта (в данном примере это файл test.data), и мы всегда имеем возможность обратиться к байтам, лежащим внутри области, ограниченной размером файла и размером отображения. Те же результаты получаются при запуске этой программы в Solaris 2.6.

В этом разделе мы работали с отображением файлов в память с помощью mmap. В упражнении 13.1 нам придется немного изменить две наших программы для работы с разделяемой памятью Posix, и мы получим те же результаты.

12.7. Резюме

Разделяемая память представляет собой самую быстродействующую форму IPC, потому что данные из разделяемой области памяти доступны всем потокам и процессам, с ней работающим. Обычно для координации совместных действий потоков и процессов, использующих разделяемую память, требуется некоторая форма синхронизации.

В этой главе мы подробно рассмотрели свойства функции `mmap` и отображение обычных файлов в память, потому что это один из способов обеспечения взаимодействия между неродственными процессами. После отображения файла в память для обращения к нему больше не нужно использовать вызовы `read`, `write` и `lseek` — вместо этого можно напрямую работать с ячейками памяти, относящимися к той области, указатель на которую возвращает `mmap`. Замена явных операций с файлом на обращение к ячейкам памяти может упростить программу и в некоторых случаях увеличить быстродействие.

Если необходимо совместное использование области памяти после вызова `fork`, можно упростить решение этой задачи, используя неименованное отображение в память. Для этого в ядрах Berkeley при вызове `mmap` указывается флаг `MAP_ANON`, а в ядрах SVR4 производится отображение специального файла `/dev/zero`.

Причина, по которой мы столь детально разобрали работу `mmap`, заключается в том, что отображение файлов в память часто оказывается очень полезным, а также в том, что `mmap` используется для работы с разделяемой памятью Posix, которая является предметом изучения следующей главы.

Для работы с памятью стандартом Posix определено еще четыре функции:

- `mlockall` делает всю память процесса резидентной; `munlockall` снимает эту блокировку;
- `mlock` делает определенный диапазон адресов процесса резидентным. Аргументами функции являются начальный адрес и длина области. Функция `munlock` разблокирует указываемую область памяти.

Упражнения

1. Что произойдет с программой в листинге 12.7, если добавить еще один повтор цикла `for`?
2. Предположим, что имеются два процесса, один из которых отправляет сообщения другому. Для этого используются очереди сообщений System V. Нарисуйте схему передачи сообщений от отправителя к получателю. Теперь представьте, что используется реализация очередей сообщений Posix из раздела 5.8, и нарисуйте аналогичную схему.
3. Мы говорили, что при вызове `mmap` с флагом `MAP_SHARED` для синхронизации содержимого файла и памяти используются алгоритмы ядра для работы с виртуальной памятью. Прочтите страницу документации, относящуюся к `/dev/zero`, чтобы узнать, что происходит, когда ядро записывает изменения обратно в этот файл.
4. Измените программу в листинге 12.2, указав `MAP_PRIVATE` вместо `MAP_SHARED`. Проверьте, что результаты будут такими же, как и при выполнении программы из листинга 12.1. Что будет содержаться в файле, отображенном в память?
5. В разделе 6.9 мы отметили, что единственным способом использовать `select` с очередью сообщений System V является создание неименованной области памяти, порождение процесса и блокирование его в вызове `msggrcv`, причем сообщение должно считываться в разделяемую память. Родительский процесс также создает два канала, один из которых используется для уведомления его о том, что сообщение помещено в разделяемую память, а другой — для уведомления дочернего процесса о возможности помещения нового сообщения в эту память. Тогда родительский процесс может вызвать `select` для открытого на чтение конца канала вместе с любыми другими дескрипторами. Напишите программу, реализующую этот алгоритм. Для выделения области неименованной разделяемой памяти используйте функцию `my_shm` (листинг A.31). Для создания очереди сообщений и помещения в нее записей используйте программы `msgcreate` и `msgsnd` из раздела 6.6. Родительский процесс должен просто выводить размер и тип всех считываемых дочерним процессом сообщений.

13.1. Введение

В предыдущей главе рассматривались общие вопросы, связанные с разделяемой памятью, и детально разбиралась функция `mmap`. Были приведены примеры, в которых вызов `mmap` использовался для создания области памяти, совместно используемой родительским и дочерним процессами. В этих примерах использовалось:

- отображение файлов в память (листинг 12.2);
- неименованное отображение памяти в системе 4.4BSD (листинг 12.4);
- неименованное отображение файла `/dev/zero` (листинг 12.5).

Теперь мы можем расширить понятие разделяемой памяти, включив в него память, совместно используемую неродственными процессами. Стандарт Posix.1 предоставляет два механизма совместного использования областей памяти для неродственных процессов:

1. Отображение файлов в память: файл открывается вызовом `open`, а его дескриптор используется при вызове `mmap` для отображения содержимого файла в адресное пространство процесса. Этот метод был описан в главе 12, и его использование было проиллюстрировано на примере родственных процессов. Однако он позволяет реализовать совместное использование памяти и для неродственных процессов.

2. Объекты разделяемой памяти: функция `shm_open` открывает объект IPC с именем стандарта Posix (например, полным именем объекта файловой системы), возвращая дескриптор, который может быть использован для отображения в адресное пространство процесса вызовом `mmap`. Данный метод будет описан в этой главе.

Оба метода требуют вызова `mmap`. Отличие состоит в методе получения дескриптора, являющегося аргументом `mmap`: в первом случае он возвращается функцией `open`, а во втором — `shm_open`. Мы показываем это на рис. 13.1. Стандарт Posix называет *объектами памяти* (memory objects) и отображеные в память файлы, и объекты разделяемой памяти стандарта Posix.

13.2. Функции `shm_open` и `shm_unlink`

Процесс получения доступа к объекту разделяемой памяти Posix выполняется в два этапа:

1. Вызов `shm_open` с именем IPC в качестве аргумента позволяет либо создать новый объект разделяемой памяти, либо открыть существующий.



Рис. 13.1. Объекты памяти Posix: отображаемые в память файлы и объекты разделяемой памяти

2. Вызов `mmap` позволяет отобразить разделяемую память в адресное пространство вызвавшего процесса.

Аргумент `name`, указанный при первом вызове `shm_open`, должен впоследствии использоваться всеми прочими процессами, желающими получить доступ к данной области памяти.

ПРИМЕЧАНИЕ

Причина, по которой этот процесс выполняется в два этапа вместо одного, на котором в ответ на имя объекта возвращался бы адрес соответствующей области памяти, заключается в том, что функция `mmap` уже существовала, когда эта форма разделяемой памяти была включена в стандарт Posix. Разумеется, эти два действия могли бы выполняться и одной функцией. Функция `shm_open` возвращает дескриптор (вспомните, что `mq_open` возвращает значение типа `mqd_t`, а `sem_open` возвращает указатель на значение типа `sem_t`), потому что для отображения объекта в адресное пространство процесса функция `mmap` использует именно дескриптор этого объекта.

Требования и правила, используемые при формировании аргумента `name`, были описаны в разделе 2.2.

Аргумент `oflag` должен содержать флаг `O_RDONLY` либо `O_RDWR` и один из следующих: `O_CREAT`, `O_EXCL`, `O_TRUNC`. Флаги `O_CREAT` и `O_EXCL` были описаны в разделе 2.3. Если вместе с флагом `O_RDWR` указан флаг `O_TRUNC`, существующий объект разделяемой памяти будет укорочен до нулевой длины.

Аргумент `mode` задает биты разрешений доступа (табл. 2.3) и используется только при указании флага `O_CREAT`. Обратите внимание, что в отличие от функций `mq_open` и `sem_open` для `shm_open` аргумент `mode` указывается всегда. Если флаг `O_CREAT` не указан, значение аргумента `mode` может быть нулевым.

Возвращаемое значение `shm_open` представляет собой целочисленный дескриптор, который может использоваться при вызове `mmap` в качестве пятого аргумента.

Функция `shm_unlink` удаляет имя объекта разделяемой памяти. Как и другие подобные функции (удаление файла из файловой системы, удаление очереди сообщений и именованного семафора Posix), она не выполняет никаких действий до тех пор, пока объект не будет закрыт всеми открывшими его процессами. Однако после вызова `shm_unlink` последующие вызовы `open`, `mq_open` и `sem_open` выполнятся не будут.

13.3. Функции ftruncate и fstat

Размер файла или объекта разделяемой памяти можно изменить вызовом `ftruncate`:

Стандарт Posix делает некоторые различия в определении действия этой функции для обычных файлов и для объектов разделяемой памяти.

1. Для обычного файла: если размер файла превышает значение `length`, избыточные данные отбрасываются. Если размер файла оказывается меньше значения `length`, действие функции не определено. Поэтому для переносимости следует использовать следующий способ увеличения длины обычного файла: вызов 1 `seek` со сдвигом `length-1` и запись 1 байта в файл. К счастью, почти все реализации Unix поддерживают увеличение размера файла вызовом `ftruncate`.
2. Для объекта разделяемой памяти: `ftruncate` устанавливает размер объекта равным значению аргумента `length`.

Итак, мы вызываем `ftruncate` для установки размера только что созданного объекта разделяемой памяти или изменения размера существующего объекта. При открытии существующего объекта разделяемой памяти следует воспользоваться `fstat` для получения информации о нем:

В структуре `stat` содержится больше десятка полей (они подробно описаны в главе 4 [21]), но только четыре из них содержат актуальную информацию, если `fd` представляет собой дескриптор области разделяемой памяти:

Пример использования этих двух функций будет приведен в следующем разделе.

ПРИМЕЧАНИЕ

К сожалению, стандарт Posix никак не оговаривает начальное содержимое разделяемой памяти. Описание функции `shm_open` гласит, что «объект разделяемой памяти будет иметь нулевой размер». Описание `ftruncate` гласит, что для обычных файлов (не объектов разделяемой памяти) «при увеличении размера файла он будет дополнен нулями». Однако в этом описании ничего не говорится о содержимом разделяемой памяти. Обоснование Posix.1 (*Rationale*) говорит, что «разделяемая память при расширении дополняется нулями», но это не официальный стандарт. Когда автор попытался уточнить этот вопрос в конференции `comp.std.unix`, он узнал, что некоторые производители протестовали против введения требования на заполнение памяти нулями из-за возникающих накладных расходов. Если новая область памяти не инициализируется каким-то значением (то есть содержимое остается без изменения), это может угрожать безопасности системы.

Приведем несколько примеров программ, работающих с разделяемой памятью Posix.

Программа shmcreate

Программа shmcreate, текст которой приведен в листинге 13.1, создает объект разделяемой памяти с указанным именем и длиной.

19-22 Вызов `shm_open` создает объект разделяемой памяти. Если указан параметр `-e`, будет возвращена ошибка в том случае, если такой объект уже существует. Вызов `ftruncate` устанавливает длину (размер объекта), а `mmap` отображает его содержимое в адресное пространство процесса. Затем программа завершает работу. Поскольку разделяемая память Posix обладает живучестью ядра, объект разделяемой памяти при этом не исчезает.

Программа shmunlink

В листинге 13.2 приведен текст тривиальной программы, удаляющей имя объекта разделяемой памяти из системы.

Программа shmwrite

В листинге 13.3 приведен текст программы `shmwrite`, записывающей последовательность 0, 1, 2 254, 244, 0, 1 и т. д. в объект разделяемой памяти.

10-15 Объект разделяемой памяти открывается вызовом `shm_open`. Его размер мы узнаем с помощью `fstat`. Затем файл отображается в память вызовом `mmap`, после чего его дескриптор может быть закрыт.

16-18 Последовательность записывается в разделяемую память.

Программа shmread

Программа shmread (листинг 13.4) проверяет значения, помещенные в разделяемую память программой shmwrite.

10-15 Объект разделяемой памяти открывается только для чтения, его размер получается вызовом fstat, после чего он отображается в память с доступом только на чтение, а дескриптор закрывается.

16-19 Проверяются значения, помещенные в разделяемую память вызовом shmwrite.

Примеры

Создадим объект разделяемой памяти с именем /tmp/myshm объемом 123 456 байт в системе Digital Unix 4.0B:

Мы видим, что файл с указываемым при создании объекта разделяемой памяти именем появляется в файловой системе. Используя программу od, мы можем выяснить, что после создания файл целиком заполнен нулями (восьмеричное число 0361100 — сдвиг, соответствующий байту, следующему за последним байтом файла, — эквивалентно десятичному 123 456).

Запустим программу shmwrite и убедимся в правильности записываемых значений с помощью программы od:

Мы проверили содержимое разделяемой памяти и с помощью shmread, а затем удалили объект, запустив программу shmunlink.

Если теперь мы запустим программу shmcreate в Solaris 2.6, то увидим, что файл указанного размера создается в каталоге /tmp:

Пример

Приведем теперь пример (листинг 13.5), иллюстрирующий тот факт, что объект разделяемой памяти может отображаться в области, начинающиеся с разных адресов в разных процессах.

10-14 Создаем сегмент разделяемой памяти с именем, принимаемым в качестве аргумента командной строки. Его размер устанавливается равным размеру целого. Затем открываем файл /etc/motd.

15-30 После вызова fork и родительский, и дочерний процессы вызывают mmap дважды, но в разном порядке. Каждый процесс выводит начальный адрес каждой из областей памяти. Затем дочерний процесс ждет 5 секунд, родительский процесс помещает значение 777 в область разделяемой памяти, после чего дочерний процесс считывает и выводит это значение.

Запустив эту программу, мы убедимся, что объект разделяемой памяти начинается с разных адресов в пространствах дочернего и родительского процессов:

Несмотря на разницу в начальных адресах, родительский процесс успешно помещает значение 777 по адресу 0xeeee30000, а дочерний процесс благополучно считывает его по адресу 0xeeee20000. Указатели ptr1 в родительском и дочернем процессах указывают на одну и ту же область разделяемой памяти, хотя их значения в этих процессах различны.

13.5. Увеличение общего счетчика

Разработаем программу, аналогичную приведенной в разделе 12.3, — несколько процессов увеличивают счетчик, хранящийся в разделяемой памяти. Итак, мы помещаем счетчик в разделяемую память, а для синхронизации доступа к нему используем именованный семафор. Отличие программы из этого раздела от предыдущей состоит в том, что процессы более не являются родственными. Поскольку обращение к объектам разделяемой памяти Posix и именованным семафорам Posix осуществляется по именам, процессы, увеличивающие общий счетчик, могут не состоять в родстве. Достаточно лишь, чтобы каждый из них знал имя IPC счетчика и чтобы у каждого были соответствующие разрешения на доступ к объектам IPC (области разделяемой памяти и семафору).

В листинге 13.6 приведен текст программы-сервера, которая создает объект разделяемой памяти, затем создает и инициализирует семафор, после чего завершает работу.

13-19 Программа начинает работу с вызова `shm_unlink`, на тот случай, если объект разделяемой памяти еще существует, а затем делается вызов `shm_open`, создающий этот объект. Его размер устанавливается равным размеру структуры `shmstruct` вызовом `ftruncate`, а затем `mmap` отображает объект в наше адресное пространство. После этого дескриптор объекта закрывается.

20-22 Сначала мы вызываем `sem_unlink`, на тот случай, если семафор еще существует. Затем делается вызов `sem_open` для создания именованного семафора и инициализации его единицей. Этот семафор будет использоваться в качестве взаимного исключения всеми процессами, которые будут обращаться к объекту разделяемой памяти. После выполнения этих операций семафор закрывается.

23 Процесс завершает работу. Поскольку разделяемая память Posix обладает по крайней мере живучестью ядра, объект не прекращает существования до тех пор, пока он не будет закрыт всеми открывавшими его процессами и явно удален.

Нам приходится использовать разные имена для семафора и объекта разделяемой памяти. Нет никаких гарантий, что в данной реализации к именам Posix IPC будут добавляться какие-либо суффиксы или префиксы, указывающие тип объекта (очередь сообщений, семафор, разделяемая память). Мы видели, что в Solaris эти типы имен имеют префиксы `.MQ`, `.SEM` и `.SHM`, но в Digital Unix они префиксов не имеют.

В листинге 13.7 приведен текст программы-клиента, которая увеличивает счетчик, хранящийся в разделяемой памяти, определенное число раз, блокируя семафор для каждой операции увеличения.

15-18 Вызов `shm_open` открывает объект разделяемой памяти, который должен уже существовать (поскольку не указан флаг `O_CREAT`). Память отображается в адресное пространство процесса вызовом `mmap`, после чего дескриптор закрывается.

19 Открываем именованный семафор.

20-26 Параметр командной строки позволяет указать количество увеличений счетчика. Каждый раз мы выводим предыдущее значение счетчика вместе с идентификатором процесса, поскольку одновременно работают несколько экземпляров программы.

Запустим сначала сервер, а затем три экземпляра программы-клиента в фоновом режиме.

13.6. Отправка сообщений на сервер

Изменим наше решение задачи производителей и потребителей следующим образом. Сначала запускается сервер, создающий объект разделяемой памяти, в который клиенты записывают свои сообщения. Сервер просто выводит содержимое этих сообщений, хотя задачу можно и обобщить таким образом, чтобы он выполнял действия, аналогичные демону syslog, который описан в главе 13 [24]. Мы называем группу отправляющих сообщения процессов клиентами, потому что по отношению к нашему серверу они ими и являются, однако эти клиенты могут являться серверами по отношению к другим приложениям. Например, сервер Telnet является клиентом демона syslog, когда отправляет ему сообщения для занесения их в системный журнал.

Вместо передачи сообщений одним из описанных ранее методов (часть 2) будем хранить сообщения в разделяемой памяти. Это, разумеется, потребует какой-либо формы синхронизации действий клиентов, помещающих сообщения, и сервера, читающего их. На рис. 13.2 приведена схема приложения в целом.



Рис. 13.2. Несколько клиентов отправляют сообщения серверу через разделяемую память

Перед нами взаимодействие нескольких производителей (клиентов) и одного потребителя (сервер). Разделяемая память отображается в адресное пространство сервера и каждого из клиентов.

В листинге 13.8 приведен текст заголовочного файла cliserv2.h, в котором определена структура объекта, хранимого в разделяемой памяти.

5-8 Три семафора Posix, размещаемых в памяти, используются для того же, для чего семафоры использовались в задаче производителей и потребителей в разделе 10.6. Их имена mutex, nempty, nstored. Переменная прит хранит индекс следующего помещаемого сообщения. Поскольку одновременно работают несколько производителей, эта переменная защищена взаимным исключением и хранится в разделяемой памяти вместе со всеми остальными.

9-10 Существует вероятность того, что клиент не сможет отправить сообщение из-за отсутствия свободного места для него. Если программа-клиент представляет собой сервер для других приложений (например, сервер FTP или HTTP), она не должна блокироваться в ожидании освобождения места для сообщения. Поэтому программа-клиент будет написана таким образом, чтобы она не блокировалась, но увеличивала счетчик переполнений (noverflow). Поскольку этот счетчик также является общим для всех процессов, он также должен быть защищен взаимным исключением, чтобы его значение не было повреждено.

11-12 Массив msgoff содержит сдвиги сообщений в массиве msgdata, в котором сообщения хранятся подряд. Таким образом, сдвиг первого сообщения msgoff[0] = 0, msgoff [1] = 256 (значение MESGSIZE), msgoff [2] = 512 и т. д.

Нужно понимать, что при работе с разделяемой памятью использовать сдвиг в таких случаях необходимо, поскольку объект разделяемой памяти может быть отображен в разные области адресного пространства процесса (может начинаться с разных физических адресов). Возвращаемое mmap значение для каждого процесса может быть индивидуальным. Поэтому при работе с объектами разделяемой памяти нельзя использовать указатели, содержащие реальные адреса переменных в этом объекте.

В листинге 13.9 приведен текст программы-сервера, которая ожидает помещения сообщений в разделяемую память, а затем выводит их.

10-16 Сначала делается вызов shm_unlink, чтобы удалить объект с тем же именем, который мог остаться после другого приложения. Затем объект разделяемой памяти создается вызовом shm_open и отображается в адресное пространство процесса вызовом mmap, после чего дескриптор объекта закрывается.

17-19 Массив сдвигов инициализируется сдвигами сообщений.

20-24 Инициализируются четыре семафора, размещаемые в объекте разделяемой памяти. Второй аргумент sem_init всегда делается ненулевым, поскольку семафоры будут использоваться совместно

несколькими процессами.

25-36 Первая половина цикла for написана по стандартному алгоритму потребителя: ожидание изменения семафора nstored, установка блокировки для семафора mutex, обработка данных, увеличение значения семафора nempty.

37-43 При каждом проходе цикла мы проверяем наличие возникших переполнений. Сравнивается текущее значение noverflows с предыдущим. Если значение изменилось, оно выводится на экран и сохраняется. Обратите внимание, что значение считывается с заблокированным взаимным исключением noverflowmutex, но блокировка снимается перед сравнением и выводом значения. Идея в том, что нужно всегда следовать общему правилу минимизации количества операций, выполняемых с заблокированным взаимным исключением. В листинге 13.10 приведен текст программы-клиента.

10-13 Первый аргумент командной строки задает имя объекта разделяемой памяти; второй — количество сообщений, которые должны быть отправлены серверу данным клиентом. Последний аргумент задает паузу перед отправкой очередного сообщения (в микросекундах). Мы сможем получить ситуацию переполнения, запустив одновременно несколько экземпляров клиентов и указав небольшое значение для этой паузы. Таким образом мы сможем убедиться, что сервер корректно обрабатывает ситуацию переполнения.

14-18 Мы открываем объект разделяемой памяти, предполагая, что он уже создан и проинициализирован сервером, а затем отображаем его в адресное пространство процесса. После этого дескриптор может быть закрыт.

19-31 Клиент работает по простому алгоритму программы-производителя, но вместо вызова sem_wait(nempty), который приводил бы к блокированию клиента в случае отсутствия места в буфере для следующего сообщения, мы вызываем sem_trywait — эта функция не блокируется. Если значение семафора нулевое, возвращается ошибка EAGAIN. Мы обрабатываем эту ошибку, увеличивая значение счетчика переполнений.

ПРИМЕЧАНИЕ

sleep_us — функция из листингов C.9 и C.10 [21]. Она приостанавливает выполнение программы на заданное количество микросекунд. Реализуется вызовом select или poll.

32-37 Пока заблокирован семафор mutex, мы можем получить значение сдвига (offset) и увеличить счетчик прит, но мы снимаем блокировку с этого семафора перед операцией копирования сообщения в разделяемую память. Когда семафор заблокирован, должны выполняться только самые необходимые операции.

Сначала запустим сервер в фоновом режиме, а затем запустим один экземпляр программы-клиента, указав 50 сообщений и нулевую паузу между ними:

Но если мы запустим программу-клиент еще раз, то мы увидим возникновение переполнений.

На этот раз клиент успешно отправил сообщения 0-9, которые были получены и выведены сервером. Затем клиент снова получил управление и поместил сообщения 10-49, но места хватило только для первых 15, а последние 25 (с 25 по 49) не были сохранены из-за переполнения:

Очевидно, что в этом примере переполнение возникло из-за того, что мы потребовали от клиента отправлять сообщения так часто, как только возможно, не делая между ними пауз. В реальном мире такое случается редко. Целью этого примера было продемонстрировать обработку ситуаций, в которых места для очередного сообщения нет, но клиент не должен блокироваться. Такая ситуация может возникнуть, разумеется, не только при использовании разделяемой памяти, но и при использовании очередей сообщений, именованных и неименованных каналов.

ПРИМЕЧАНИЕ

Переполнение приемного буфера данными встречается не только в этом примере. В разделе 8.13 [24] обсуждалась такая ситуация в связи с дейтаграммами UDP и приемным буфером сокета UDP. В разделе 18.2 [23] подробно рассказывается о том, как доменные сокеты Unix возвращают отправителю ошибку ENOBUFS при переполнении приемного буфера получателя. Это отличает доменные сокеты от протокола UDP. Программа-клиент из листинга 13.10 узнает о переполнении буфера, поэтому если этот код поместить в функцию общего назначения, которую затем будут использовать другие программы, такая функция сможет возвращать ошибку вызывающему процессу при переполнении буфера сервера.

13.7. Резюме

Разделяемая память Posix реализуется с помощью функции `mmap`, обсуждавшейся в предыдущей главе. Сначала вызывается функция `shm_open` с именем объекта Posix IPC в качестве одного из аргументов. Эта функция возвращает дескриптор, который затем передается функции `mmap`. Результат аналогичен отображению файла в память, но разделяемая память Posix не обязательно реализуется через файл.

Поскольку доступ к объектам разделяемой памяти может быть получен через дескриптор, для установки размера объекта используется функция `ftruncate`, а информация о существующем объекте (биты разрешений, идентификаторы пользователя и группы, размер) возвращается функцией `fstat`.

В главах, рассказывающих об очередях сообщений и семафорах Posix, мы приводили примеры их реализации через отображение в память (разделы 5.8 и 10.15). Для разделяемой памяти Posix мы этого делать не будем, поскольку реализация тривиальна. Если мы готовы использовать отображение в файл (что и сделано в Solaris и Digital Unix), `shm_open` реализуется через `open`, а `shm_unlink` — через `unlink`.

Упражнения

1. Измените программы из листингов 12.6 и 12.7 таким образом, чтобы они работали с разделяемой памятью Posix, а не с отображаемым в память файлом. Убедитесь, что результаты будут такими же, как и для отображаемого в память файла.
2. В циклах for в листингах 13.3 и 13.4 используется команда `*ptr++` для перебора элементов массива. Не лучше ли было использовать `ptr[i]`?

14.1. Введение

Основные принципы разделяемой памяти System V совпадают с концепцией разделяемой памяти Posix. Вместо вызовов `shm_open` и `mmap` в этой системе используются вызовы `shmget` и `shmat`.

Для каждого сегмента разделяемой памяти ядро хранит нижеследующую структуру, определенную в заголовочном файле `<sys/shm.h>`:

Структура `ipc_perm` была описана в разделе 3.3; она содержит разрешения доступа к сегменту разделяемой памяти.

14.2. Функция `shmget`

С помощью функции `shmget` можно создать новый сегмент разделяемой памяти или подключиться к существующему:

Возвращаемое целочисленное значение называется идентификатором разделяемой памяти. Он используется с тремя другими функциями `shmXXX`.

Аргумент *key* может содержать значение, возвращаемое функцией `ftok`, или константу `IPC_PRIVATE`, как обсуждалось в разделе 3.2.

Аргумент *size* указывает размер сегмента в байтах. При создании нового сегмента разделяемой памяти нужно указать ненулевой размер. Если производится обращение к существующему сегменту, аргумент *size* должен быть нулевым.

Флаг *oflag* представляет собой комбинацию флагов доступа на чтение и запись из табл. 3.3. К ним могут быть добавлены с помощью логического сложения флаги `IPC_CREAT` или `IPC_CREAT | IPC_EXCL`, как уже говорилось в связи с рис. 3.2.

Новый сегмент инициализируется нулями.

Обратите внимание, что функция `shmget` создает или открывает сегмент разделяемой памяти, но не дает вызвавшему процессу доступа к нему. Для подключения сегмента памяти предназначена функция `shmat`, описанная в следующем разделе.

14.3. Функция shmat

После создания или открытия сегмента разделяемой памяти вызовом `shmget` его нужно подключить к адресному пространству процесса вызовом `shmat`:

Аргумент `shmid` — это идентификатор разделяемой памяти, возвращенный `shmget`. Функция `shmat` возвращает адрес начала области разделяемой памяти в адресном пространстве вызвавшего процесса. Правила, по которым формируется этот адрес, таковы:

- если аргумент `shmaddr` представляет собой нулевой указатель, система сама выбирает начальный адрес для вызвавшего процесса. Это рекомендуемый (и обеспечивающий наилучшую совместимость) метод;
- если `shmaddr` отличен от нуля, возвращаемый адрес зависит от того, был ли указан флаг `SHM_RND` (в аргументе `flag`):
 - если флаг `SHM_RND` не указан, разделяемая память подключается непосредственно с адреса, указанного аргументом `shmaddr`,
 - если флаг `SHM_RND` указан, сегмент разделяемой памяти подключается с адреса, указанного аргументом `shmaddr`, округленного вниз до кратного константе `SHMLBA`. Аббревиатура `LBA` означает `lower boundary address` — нижний граничный адрес.

По умолчанию сегмент подключается для чтения и записи, если процесс обладает соответствующими разрешениями. В аргументе `flag` можно указать константу `SHM_RDONLY`, которая позволит установить доступ только на чтение.

14.4. Функция shmdt

После завершения работы с сегментом разделяемой памяти его следует отключить вызовом shmdt:

При завершении работы процесса все сегменты, которые не были отключены им явно, отключаются автоматически.

Обратите внимание, что эта функция не удаляет сегмент разделяемой памяти. Удаление осуществляется функцией shmctl с командой IPC_RMIO.

14.5. Функция shmctl

Функция `shmctl` позволяет выполнять различные операции с сегментом разделяемой памяти:

Команд (значений аргумента *cmd*) может быть три:

- `IPC_RMID` — удаление сегмента разделяемой памяти с идентификатором *shmid* из системы;
- `IPC_SET` — установка значений полей структуры `shmid_ds` для сегмента разделяемой памяти равными значениям соответствующих полей структуры, на которую указывает аргумент *buff*: `shm_perm.uid`, `shm_perm.gid`, `shm_perm.mode`. Значение поля `shm_ctime` устанавливается равным текущему системному времени;
- `IPC_STAT` — возвращает вызывающему процессу (через аргумент *buff*) текущее значение структуры `shmid_ds` для указанного сегмента разделяемой памяти.

В этом разделе приведено несколько примеров простых программ, иллюстрирующих работу с разделяемой памятью System V.

Программа shmget

Программа `shmget`, текст которой приведен в листинге 14.1, создает сегмент разделяемой памяти, принимая из командной строки полное имя и длину сегмента.

19 Вызов `shmget` создает сегмент разделяемой памяти указанного размера. Полное имя, передаваемое в качестве аргумента командной строки, преобразуется в ключ IPC System V вызовом `ftok`. Если указан параметр `-e`, наличие существующего сегмента с тем же именем приведет к возвращению ошибки. Если мы знаем, что сегмент уже существует, в командной строке должна быть указана нулевая длина.

20 Вызов `shmat` подключает сегмент к адресному пространству процесса. После этого программа завершает работу. Разделяемая память System V обладает поменьшей мере живучестью ядра, поэтому сегмент разделяемой памяти при этом не удаляется.

Программа shmrmid

В листинге 14.2 приведен текст тривиальной программы shmrmid, которая вызывает shmctl с командой IPC_RMID для удаления сегмента разделяемой памяти из системы.

Программа shmwrite

В листинге 14.3 приведен текст программы `shmwrite`, которая заполняет сегмент разделяемой памяти последовательностью значений 0, 1, 2, ..., 254, 255, 0, 1 и т. д.

10-12 Сегмент разделяемой памяти открывается вызовом `shmget` и подключается вызовом `shmat`. Его размер может быть получен вызовом `shmctl` с командой `IPC_STAT`.

13-15 В разделяемую память записывается последовательность значений.

Программа shmread

Программа shmread, текст которой приведен в листинге 14.4, проверяет последовательность значений, записанную в разделяемую память программой shmwrite.

10-12 Открываем и подключаем сегмент разделяемой памяти. Его размер может быть получен вызовом shmctl с командой IPC_STAT. 13-16 Проверяется последовательность, записанная программой shmwrite.

Примеры

Создадим сегмент разделяемой памяти длиной 1234 байта в системе Solaris 2.6. Для идентификации сегмента используем полное имя нашего исполняемого файла `shmget`. Это имя будет передано функции `ftok`. Имя исполняемого файла сервера часто используется в качестве уникального идентификатора для данного приложения:

Программу `ipcs` мы запускаем для того, чтобы убедиться, что сегмент разделяемой памяти действительно был создан и не был удален по завершении программы `shmcreate`. Количество подключений (хранящееся в поле `shm_nattch` структуры `shmid_ds`) равно нулю, как мы и предполагали.

Теперь запустим программу `shmwrite`, чтобы заполнить содержимое разделяемой памяти последовательностью значений. Затем проверим содержимое сегмента разделяемой памяти программой `shmread` и удалим этот сегмент:

Мы используем программу `ipcs`, чтобы убедиться, что сегмент разделяемой памяти действительно был удален.

ПРИМЕЧАНИЕ

При использовании имени исполняемого файла сервера в качестве аргумента `ftok` для идентификации какого-либо вида IPC System V обычно передается полное имя этого файла (например, `/usr/bin/myserverd`), а не часть имени, как сделано у нас (`shmget`). У нас не возникло проблем в этом примере, потому что все программы запускались из того же каталога, в котором был расположен исполняемый файл сервера. Вы помните, что функция `ftok` использует номер i-node файла для формирования ключа IPC и ей безразлично, определяется файл своим полным именем или его частью (относительным именем).

На разделяемую память System V накладываются определенные ограничения точно так же, как и на семафоры и очереди сообщений System V (раздел 3.8). В табл. 14.1 приведены значения этих ограничений для разных реализаций. В первом столбце приведены традиционные для System V имена переменных ядра, в которых хранятся эти ограничения.

Таблица 14.1. Типичные значения ограничений, накладываемых на разделяемую память System V

Пример

Программа в листинге 14.5 определяет значения четырех ограничений, приведенных в табл. 14.1.

Запустив эту программу в Digital Unix 4.0B, увидим:

Причина, по которой в табл. 14.1 приведено значение 128 для числа идентификаторов, а наша программа выводит значение 127, заключается в том, что один сегмент разделяемой памяти всегда используется системным демоном.

14.8. Резюме

Разделяемая память System V похожа на разделяемую память Posix. Наиболее схожи функции:

- `shmget` для получения идентификатора;
- `shmat` для подключения сегмента разделяемой памяти к адресному пространству процесса;
- `shmctl` с командой `IPC_STAT` для получения размера существующего сегмента разделяемой памяти;
- `shmctl` с командой `IPC_RMID` для удаления объекта разделяемой памяти.

Одно из отличий состоит в том, что размер объекта разделяемой памяти Posix может быть изменен в любой момент вызовом `ftruncate` (как мы продемонстрировали в упражнении 13.1), тогда как размер объекта разделяемой памяти System V устанавливается изначально вызовом `shmget` и не может быть изменен.

Упражнение

Листинг 6.6 содержал измененную версию программы из листинга 6.4. Новая программа использовала для обращения к объекту IPC System V идентификатор вместо полного имени. Таким образом мы показали, что для доступа к очереди сообщений System V достаточно знать только ее идентификатор (если у нас имеются соответствующие разрешения). Сделайте аналогичные изменения в программе из листинга 14.4, продемонстрировав, что это верно и для разделяемой памяти System V.

Поговорим о схеме клиент-сервер и вызове процедур. Существуют три различных типа вызова процедур, показанные на рис. 15.1.



Рис. 15.1. Три типа вызова процедур

1. Локальный вызов процедуры (local procedure call) знаком нам по обычному программированию на языке C. Вызываемая и вызывающая процедуры (функции) при этом относятся к одному и тому же процессу. При этом обычно выполняется некая команда процессора, передающая управление новой процедуре, а вызвавшая процедура сохраняет значение регистров процессора и выделяет место в стеке под свои локальные переменные.

2. Удаленный вызов процедуры (remote procedure call — RPC) происходит в ситуации, когда вызвавшая и вызываемая процедуры относятся к разным процессам. В такой ситуации мы обычно называем вызвавшую процедуру клиентом, а вызванную — сервером. Во втором сценарии на рис. 15.1 клиент и сервер выполняются на одном и том же узле. Это типичный частный случай третьего сценария, и это именно то, что осуществляется с помощью дверей (doors). Итак, двери дают возможность вызывать процедуру (функцию) другого процесса на том же узле. Один из процессов (сервер) делает процедуру, находящуюся внутри него, доступной для вызова другим процессам (клиентам), создавая для этой процедуры дверь. Мы можем считать двери специальным типом IPC, поскольку при этом между процессами (клиентом и сервером) передается информация в форме аргументов функции и возвращаемых значений.

3. RPC в общем случае дает возможность клиенту на одном узле вызвать процедуру сервера на другом узле, если эти два узла связаны каким-либо образом по сети (третий сценарий на рис. 15.1). Такой вид взаимодействия будет описан в главе 16.

ПРИМЕЧАНИЕ

Впервые двери были разработаны для распределенной операционной системы Spring. Детали этого проекта доступны по адресу <http://www.sun.com/tech/projects/spring>. Описание механизма дверей в этой операционной системе можно найти в книге [7].

Затем двери появились в версии Solaris 2.5, хотя единственная страница документации, к ним относящаяся, содержала только предупреждение о том, что двери являются экспериментальным интерфейсом, используемым отдельными приложениями Sun. В Solaris 2.6 описание этого интерфейса занимает уже 8 страниц, но в них он характеризуется как «развивающийся». В будущих версиях Solaris 2.6 описываемый в этой главе интерфейс API может быть изменен. Предварительная версия дверей для Linux уже разрабатывается, детали можно выяснить по адресу <http://www.cs.brown.edu/~tor/doors>.

Чтобы воспользоваться интерфейсом дверей в Solaris 2.6, нужно подключить соответствующую библиотеку (-ldoor), содержащую функции door_XXX, описываемые в этой главе, и использовать файловую систему ядра (/kernel/sys/doorfs).

Хотя двери поддерживаются только в системе Solaris, мы описываем их достаточно подробно, поскольку это описание позволяет подготовить читателя к удаленному вызову процедур без необходимости обсуждать какие-либо детали сетевого интерфейса. В приложении А мы увидим, что этот интерфейс достаточно быстр — едва ли не быстрее, чем все остальные средства передачи сообщений.

Локальные вызовы процедур являются синхронными (synchronous): вызывающий процесс не получает управление до тех пор, пока не происходит возврат из вызванной процедуры. Потоки могут восприниматься как средство асинхронного вызова процедур: функция (указанный в третьем аргументе pthread_create) выполняется одновременно с вызвавшим процессом. Вызвавший процесс может ожидать завершения вызванного процесса с помощью функции pthread_join. Удаленный вызов процедур может быть как синхронным, так и асинхронным, но мы увидим, что вызовы через двери являются синхронными.

Внутри процесса двери идентифицируются дескрипторами. Извне двери могут идентифицироваться именами в файловой системе. Сервер создает дверь вызовом door_create; аргументом этой функции является указатель на процедуру, которая будет связана с данной дверью, а возвращаемое значение является дескриптором двери. Затем сервер связывает полное имя файла с дескриптором двери с помощью функции fattach. Клиент открывает дверь вызовом open, при этом аргументом функции является полное имя файла, которое сервер связал с дверью, а возвращаемым значением — дескриптор, который будет использоваться клиентом для доступа к двери. Затем клиент может вызывать процедуру с помощью door_call. Естественно, программа, являющаяся сервером для

некоторой двери, может являться клиентом для другой.

Мы уже сказали, что вызовы через двери являются синхронными: когда клиент вызывает `door_call`, возврата из этой функции не происходит до тех пор, пока процедура на сервере не завершит работу (возможно, с ошибкой). Реализация дверей в Solaris связана с потоками. Каждый раз, когда клиент вызывает процедуру сервера, для обработки этого вызова создается новый поток в процессе-сервере. Работа с потоками обычно осуществляется автоматически функциями библиотеки дверей, при этом потоки создаются по мере необходимости, но мы увидим, что сервер может и сам управлять этими потоками, если это требуется. Это также означает, что одна и та же процедура может выполняться сервером для нескольких клиентов, причем для каждого из них будет создан отдельный поток. Такой сервер является параллельным. Поскольку одновременно могут выполняться несколько экземпляров процедуры сервера (каждая из которых представляет собой отдельный поток), содержимое этих процедур должно соответствовать определенным требованиям, которые обычно предъявляются к многопоточным программам.

При удаленном вызове процедуры и данные, и дескрипторы могут быть переданы от клиента к серверу. Обратно также могут быть переданы данные и дескрипторы. Передача дескрипторов вообще является неотъемлемым свойством дверей. Более того, поскольку двери идентифицируются дескрипторами, это позволяет процессу передать дверь другому процессу. Более подробно о передаче дескрипторов будет говориться в разделе 15.8.

Пример

Начнем описание интерфейса дверей с простого примера: клиент передает серверу длинное целое, а сервер возвращает клиенту квадрат этого значения тоже как длинное целое. В листинге 15.1 приведен текст программы-клиента (в этом примере мы опускаем множество деталей, большую часть которых мы обсудим далее в тексте главы).

8-10 Дверь задается полным именем, передаваемым в качестве аргумента командной строки. Она открывается вызовом open. Возвращаемый дескриптор называется дескриптором двери, но часто его самого и называют дверью.

11-18 Структура arg содержит указатели на аргументы и результат. Поле data_ptr указывает на первый байт аргументов, а data_size содержит количество байтов в аргументах. Два поля desc_ptr и desc_num предназначены для передачи дескрипторов, о чем мы будем подробно говорить в разделе 15.8. rbuf указывает на первый байт буфера результата, а rsize задает его размер.

19-21 Мы вызываем процедуру на сервере с помощью door_call; аргументами этого вызова являются дескриптор двери и указатель на структуру аргументов. После возвращения из этого вызова программа печатает получившийся результат.

Программа-сервер приведена в листинге 15.2. Она состоит из процедуры сервера с именем servproc и функции main.

2-10 Процедура сервера вызывается с пятью аргументами, но мы используем только один из них — dataptr. Он указывает на первый байт аргумента. Аргумент, представляющий собой длинное целое, передается через этот указатель и возводится в квадрат. Управление передается клиенту вместе с результатом вызовом door_return. Первый аргумент указывает на результат, второй задает его размер, а оставшиеся предназначены для возврата дескрипторов.

17-21 Дескриптор двери создается вызовом door_create. Первый аргумент является указателем на функцию, соответствующую этой двери (segvrgos). После получения этого дескриптора его нужно связать с некоторым именем в файловой системе, поскольку оно будет использоваться клиентом для подключения к этой двери. Делается это путем создания обычного файла в файловой системе (сначала мы вызываем unlink, на тот случай, если такой файл уже существует, причём возможная ошибка игнорируется) и вызова fattach — функции SVR4, связывающей дескриптор с полным именем файла.

22-24 Главный поток сервера блокируется при вызове pause. Вся функциональность обеспечивается функцией segvrgos, которая будет запускаться как отдельный поток каждый раз при получении запроса клиента.

Запустим сервер в отдельном окне:

После этого запустим программу-клиент в другом окне, указав в качестве аргумента то же полное имя, которое было указано при вызове сервера:

Мы получили ожидаемый результат. Вызвав ls, мы видим, что эта программа выводит букву D в начале строки, соответствующей файлу, указывая, что этот файл является дверью.

На рис. 15.2 приведена диаграмма работы данного примера. Функция door_call вызывает процедуру на сервере, которая затем вызывает door_return для возврата.

На рис. 15.3 приведена диаграмма, показывающая, что в действительности происходит при вызове процедуры в другом процессе на том же узле.



Рис. 15.2. Внешний вид вызова процедуры в другом процессе

На рис. 15.3 выполняются следующие действия:

0. Запускается сервер, вызывает door_create, чтобы создать дескриптор для функции servproc, затем связывает этот дескриптор с именем файла в файловой системе.

1. Запускается клиент и вызывает `door_call`. Это функция в библиотеке дверей.
2. Библиотечная функция `door_call` делает системный вызов. При этом указывается процедура, которая должна быть выполнена, а управление передается функции из библиотеки дверей процесса-сервера.
3. Вызывается процедура сервера (`servproc` в данном примере).
4. Процедура сервера делает все необходимое для обработки запроса клиента и вызывает `door_return` по завершении работы.
5. Библиотечная функция `door_return` осуществляет системный вызов, передавая управление ядру.
6. В этом вызове указывается процесс-клиент, которому и передается управление.



Рис. 15.3. Что в действительности происходит при вызове процедуры в другом процессе

Последующие разделы этой главы описывают интерфейс дверей (doors API) более подробно, с множеством примеров. В приложении А мы убедимся, что двери представляют собой наиболее быструю форму IPC (при измерении времени ожидания).

15.2. Функция door_call

Функция `door_call` вызывается клиентом для обращения к процедуре сервера, выполняемой в адресном пространстве процесса-сервера:

Дескриптор `fd` обычно возвращается функцией `open` (см. листинг 15.1). Полное имя файла, открываемого клиентом, однозначно идентифицирует процедуру сервера, которая вызывается `door_call` при передаче дескриптора.

Второй аргумент — `argp` — указывает на структуру, описывающую аргументы и приемный буфер для возвращаемых значений:

При возврате из удаленного вызова эта структура описывает возвращаемые значения. Все поля структуры могут быть изменены при возврате, мы подробно рассмотрим это ниже.

ПРИМЕЧАНИЕ

Использование типа `char*` для двух указателей кажется странным и требует использования явного преобразования типов для предотвращения вывода предупреждений компилятора. Естественно было бы использовать указатели типа `void*`. С указателями `char*` мы еще столкнемся в функции `door_return`. Вероятно, в Solaris 2.7 тип данных `desc_num` изменится на `unsigned int` и последний аргумент `door_return` изменится соответствующим образом.

Аргументы и результаты могут быть двух типов: данные и дескрипторы.

■ Аргументы-данные представляют собой последовательность данных длиной `data_size` байт. На эту последовательность должен указывать `data_ptr`. Клиент и сервер должны заранее знать формат этих данных (и аргументов, и результатов). Нет способа указать серверу тип аргументов. В программах листингов 15.1 и 15.2 клиент и сервер были написаны таким образом, что они оба знали, что аргумент представлял собой одно длинное целое и возвращаемый результат также был одним длинным целым. Для скрытия внутреннего устройства передаваемых данных их можно объединить в структуру, что упростит работу тому, кто будет читать код несколько лет спустя. Итак, все аргументы можно заключить в одну структуру, результаты — в другую и обе их определить в одном заголовочном файле, используемом клиентом и сервером. Пример будет приведен в листингах 15.8 и 15.9. Если аргументов-данных нет, указатель `data_ptr` должен быть нулевым и размер данных `data_size` должен иметь значение 0.

ПРИМЕЧАНИЕ

Поскольку клиент и сервер работают с аргументами и результатами, помещаемыми в соответствующие буфера, компилировать их следует одним и тем же компилятором, поскольку разные компиляторы могут по-разному упорядочивать данные в структурах даже в одной и той же системе.

■ Аргументы-дескрипторы хранятся в массиве структур `door_desc_t`, каждая из которых содержит один передаваемый от клиента серверу дескриптор. Количество структур типа `door_desc_t` задается аргументом `desc_num`. (Мы описываем эту структуру и смысл «передачи дескриптора» в разделе 15.8.) Если аргументов-дескрипторов нет, следует передать нулевой указатель `desc_ptr` и присвоить полю `desc_num` значение 0.

■ При возврате из функции `data_ptr` указывает на результаты-данные, а `data_size` задает размер возвращаемых данных. Если никакие данные не возвращаются, `data_size` будет иметь значение 0, а значение указателя `data_ptr` следует игнорировать.

■ Функция может возвращать и дескрипторы, при этом `desc_ptr` указывает на массив структур типа `door_desc_t`, каждая из которых содержит один передаваемый сервером клиенту дескриптор. Количество возвращаемых структур типа `door_desc_t` хранится в поле `desc_num`. Если дескрипторы не возвращаются, значение `desc_num` будет равно 0, а указатель `desc_ptr` следует игнорировать.

Можно спокойно использовать один и тот же буфер для передаваемых аргументов и возвращаемых результатов. При вызове `door_call` и `data_ptr`, и `desc_ptr` могут указывать на буфер, указанный аргументом `rbuf`.

Перед вызовом `door_call` клиент устанавливает указатель `rbuf` на буфер для результатов, а `rsizе` делает равным размеру буфера. После возвращения из функции и `data_ptr`, и `desc_ptr` будут указывать на этот буфер. Если он слишком мал для хранения результатов, возвращаемых сервером, библиотека дверей автоматически выделит новый буфер в адресном пространстве клиента с помощью `mmap` (раздел 12.2) и обновит значения `rbuf` и `rsizе` соответствующим образом. Поля `data_ptr` и `desc_ptr` будут указывать на новый буфер. Клиент отвечает за то, чтобы обнаружить изменение этих указателей и впоследствии освободить занимаемую память вызовом `munmap` с

аргументами rbuf и rsize. Пример будет приведен в листинге 15.4.

15.3. Функция door_create

Процесс-сервер определяет некоторую функцию как процедуру сервера вызовом door_create:

Здесь мы добавили наше собственное определение типа, что упрощает прототип функции. Это определение утверждает, что процедуры сервера (например, servgros в листинге 15.2) вызываются с пятью аргументами и ничего не возвращают.

Когда сервер вызывает door_create, первый аргумент (*proc*) указывает адрес процедуры сервера, которая будет вызываться через дескриптор двери, возвращаемый этим вызовом. При вызове процедуры сервера ее аргумент *cookie* содержит значение, передаваемое в качестве второго аргумента door_create. Это дает серверу возможность передавать процедуре какой-либо указатель каждый раз, когда эта процедура вызывается клиентом. Следующих четыре аргумента процедуры сервера — *dataptr*, *datasize*, *descptr* и *ndesc* — описывают аргументы-данные и аргументы-дескрипторы клиента. Они соответствуют первым четырем полям структуры door_arg_t, описанной в предыдущем разделе.

Последний аргумент door_create(*attr*) описывает специальные атрибуты процедуры сервера и может быть равен либо 0, либо логической сумме двух констант:

■ DOOR_PRIVATE — библиотека дверей автоматически создает новые потоки в процессе-сервере при поступлении запросов от клиентов. По умолчанию эти потоки помещаются в пул потоков и могут использоваться для обслуживания запросов клиентов по всем дверям данного процесса.

Указание атрибута DOOR_PRIVATE говорит библиотеке, что для данной двери следует создать собственный пул потоков, отдельный от пула потоков процесса.

■ DOOR_UNREF — когда количество дескрипторов, открытых для данной двери, изменяется с двух до одного, процедура сервера вызывается со вторым аргументом, имеющим значение DOOR_UNREF_DATA. При этом аргумент descptr представляет собой нулевой указатель, а аргументы datasize и ndesc равны нулю. Мы приведем пример использования этого атрибута в листинге 15.13.

Возвращаемое сервером значение имеет тип void, поскольку процедура сервера никогда не завершает работу вызовом return. Вместо этого процедура должна вызывать door_return (функция описана в следующем разделе).

В листинге 15.2 мы видели, что после получения дескриптора двери вызовом door_create сервер должен вызвать fattach для связывания этого дескриптора с некоторым файлом. Клиент затем может открыть этот файл для получения дескриптора двери, который впоследствии может быть использован при вызове door_call.

ПРИМЕЧАНИЕ

Функция fattach не включена в стандарт Posix.1, но ее наличие требуется стандартом Unix 98. Кроме того, этот стандарт определяет также функцию fdetach, отключающую связь дескриптора и файла, и программу fdetach, вызывающую эту функцию.

Для дескрипторов дверей, создаваемых door_create, устанавливается бит FD_CLOEXEC. Это означает, что дескриптор закрывается при вызове процессом функций типа exec. Что касается вызова fork, несмотря на то что открытые родительским процессом дескрипторы используются дочерним процессом совместно с ним, только родительский процесс будет принимать вызовы от клиентов. Дочерним процессам вызовы не передаются, хотя дескриптор, возвращаемый door_create, и будет в них открыт.

ПРИМЕЧАНИЕ

Если мы учтем, что дверь идентифицируется с помощью PID и адреса процедуры сервера (что мы узнаем из структуры door_info_t в разделе 15.6), ограничения на вызовы exec и fork станут понятны. Дочерний процесс не будет принимать вызовов, поскольку его идентификатор процесса отличается от идентификатора, связанного с дверью. Дескриптор должен быть закрыт при вызове exec, потому что хотя идентификатор при этом и не меняется, адрес процедуры сервера уже не будет иметь никакого смысла в той программе, которая будет запущена после вызова exec.

15.4. Функция door_return

После завершения работы процедуры сервера возврат из нее осуществляется вызовом `door_return`. Это приводит к возврату из `door_call` соответствующего клиента.

Возвращаемые данные задаются аргументами `dataptr` и `datasize`, а возвращаемые дескрипторы — `descptr` и `ndesc`.

15.5. Функция door_cred

Интерфейс дверей предусматривает полезную возможность получения информации о клиенте при каждом вызове. Это осуществляется функцией `door_cred`:

Структура, на которую указывает аргумент `cred`, имеет тип `door_cred_t`, определяемый как

В эту структуру помещается информация о клиенте при возвращении из вызова `door_cred`. В разделе 4.4 [21] подробно рассказывается о различиях между действующими и реальными идентификаторами пользователя и группы, а пример использования этой функции приведен в листинге. 15.5.

Обратите внимание, что эта функция не принимает никаких дескрипторов. Она возвращает информацию о клиенте, осуществившем конкретный вызов через дверь, и поэтому должна вызываться из процедуры сервера или другой функции, вызываемой из процедуры сервера.

15.6. Функция door_info

Только что описанная функция door_cred предоставляет серверу информацию о клиенте. Клиент же может получить информацию о сервере, вызвав doo_info:

Дескриптор *fd* указывает на открытую дверь. Структура типа *door_info_t*, на которую указывает *info*, после возвращения из функции содержит информацию о сервере:

Поле *di_target* содержит идентификатор процесса сервера, а *di_proc* — адрес процедуры сервера в процессе (от которого клиенту, вообще говоря, пользы мало). Указатель, передаваемый процедуре сервера в качестве первого аргумента (*cookie*), возвращается клиенту в поле *di_data*.

Текущие атрибуты двери помещаются в поле *di_attributes*, и два из них уже были описаны в разделе 15.3. Это атрибуты DOOR_PRIVATE и DOOR_UNREF. Два других атрибута называются DOOR_LOCAL (процедура является локальной для данного процесса) и DOOR_REVOKE (сервер аннулировал процедуру, связанную с этой дверью, вызвав door_revoke).

Каждой двери при создании сопоставляется уникальный в пределах системы номер, который возвращается в поле *di_uniquifier*.

Эта функция обычно вызывается клиентом для получения информации о сервере. Однако она может быть вызвана и из процедуры сервера, причем первым аргументом в этом случае должна быть константа DOOR_QUERY. Тогда функция возвратит информацию о вызвавшем потоке, то есть о данном экземпляре процедуры сервера. В этом случае адреса процедуры сервера и принимаемых аргументов (*di_proc* и *di_data*) могут представлять интерес.

В этом разделе мы приведем примеры использования пяти только что описанных функций.

Функция door_info

В листинге 15.3 приведен текст программы, открывающей дверь и вызывающей door_info для получения информации об этой двери, которая затем выводится на экран.

Сначала программа открывает файл с указанным полным именем и проверяет, что это действительно дверь. Поле st_mode структуры stat в этом случае должно содержать такое значение, что макрос S_ISDOOR будет возвращать значение «истина». Затем вызывается функция door_info.

Сначала мы укажем этой программе полное имя файла, не являющегося дверью, а затем попробуем получить информацию о двух встроенных дверях Solaris 2.6:

Команду ps мы используем для того, чтобы узнать, какая программа выполняется с идентификатором, возвращаемым door_info.

Буфер результатов слишком мал

Когда мы рассказывали о функции `door_call`, мы отметили, что если буфер результатов оказывается слишком мал, библиотека дверей осуществляет автоматическое выделение нового буфера. Сейчас мы покажем это на примере. В листинге 15.4 приведен текст новой программы-клиента, которая представляет собой измененную версию листинга 15.2.

19-22 В этой версии программы на экран выводится адрес переменной `oval`, содержимое указателя `data_ptr`, который должен указывать на возвращаемые функцией `door_call` данные, и адрес и размер приемного буфера (`rbuf` и `rsize`).

Запустим эту программу, не изменяя размер приемного буфера по сравнению с листингом 15.2. Мы ожидаем, что `data_ptr` и `rbuf` будут указывать на переменную `oval` и `rsize` будет иметь значение 4 (4 байта в буфере). И действительно, вот что мы видим:

Изменим только одну строку в листинге 15.4, уменьшив размер буфера клиента до одного байта. Новый вариант строки 18 будет иметь вид:

Запустим новую программу и увидим, что библиотека автоматически выделила место под новый буфер результатов и `data_ptr` теперь указывает на новый буфер:

Размер выделенного буфера равен 4096 байт, что совпадает с размером страницы в данной системе, который мы узнали в разделе 12.6. Этот пример показывает, что следует всегда обращаться к результатам через указатель `data_ptr`, а не через переменные, адреса которых были переданы в `rbuf`. В нашем примере к результату типа «длинное целое» следует обращаться как `*(long*)arg.data_ptr`, а не `oval` (что мы делали в листинге 15.2).

Новый буфер выделяется вызовом `mmap` и может быть возвращен системе с помощью `munmap`. Клиент может повторно использовать этот буфер при новых вызовах `door_call`.

Функция door_cred и информация о клиенте

На этот раз мы изменим нашу функцию servproc из листинга 15.3, добавив в нее вызов door_cred для получения информации о пользователе. В листинге 15.5 приведен текст новой процедуры сервера; функции main клиента и сервера не претерпевают изменений по сравнению с листингами 15.2 и 15.3.

Сначала мы запустим программу-клиент и увидим, что действующий и реальный идентификаторы клиента совпадают, как мы и предполагали. Затем мы сменим владельца исполняемого файла на привилегированного пользователя, установим бит SUID и запустим программу снова:

Если мы посмотрим, что в это время выводил сервер, то увидим следующую картину:

Действующий идентификатор пользователя при втором запуске изменился. Значение 0 означает привилегированного пользователя.

Автоматическое управление потоками сервера

Чтобы посмотреть, как осуществляется управление потоками сервера, добавим в процедуру сервера команду выдачи ее идентификатора потока. Добавим в нее также пятисекундную паузу, чтобы имитировать длительное выполнение. За это время мы сможем запустить несколько клиентов. В листинге 15.6 приведен текст новой процедуры сервера.

Здесь используется новая функция из нашей библиотеки — `pr_thread_id`. Она принимает один аргумент (указатель на идентификатор потока или нулевой указатель вместо идентификатора вызвавшего потока) и возвращает идентификатор этого потока (обычно небольшое целое число, но всегда в формате длинного целого). Процессу всегда можно сопоставить целое число — его идентификатор. Хотя мы и не знаем, к какому типу принадлежит идентификатор процесса (`int` или `long`), мы просто преобразуем значение, возвращаемое `getpid`, к типу `long` и выводим значение (листинг 9.2). Однако идентификатор потока принадлежит к типу `pthread_t`, который не обязательно является одним из целых типов. И действительно, в Solaris 2.6 идентификаторами потоков являются короткие целые, тогда как в Digital Unix используются указатели. Однако часто возникает необходимость сопоставлять потокам небольшие целые числа для задач отладки (как в данном примере). Наша библиотечная функция, текст которой приведен в листинге 15.7, решает этот вопрос.

Если в данной реализации идентификатор потока не является небольшим целым числом, функция может быть сложнее. Она может осуществлять отображение значений типа `pthread_t` в целые числа и сохранять эти отображения для последующих вызовов в массиве или связном списке. Эта задача решена в функции `thread_name` в книге [13].

Вернемся к программе из листинга 15.6. Запустим ее три раза подряд. Поскольку нам приходится ждать возвращения подсказки интерпретатора, чтобы запустить клиент еще раз, мы можем быть уверены, что каждый раз выполняется пятисекундная пауза:

Взглянув на текст, выводимый сервером, мы увидим, что клиенты каждый раз обслуживались одним и тем же потоком сервера:

Теперь запустим три экземпляра программы-клиента одновременно:

Выводимый сервером текст показывает, что для обработки второго и третьего вызова процедуры сервера создаются новые потоки:

Затем мы запустим еще два клиента одновременно (первые три уже завершили работу):

При этом сервер использует созданные ранее потоки:

Этот пример показывает, что серверный процесс (то есть библиотека дверей, подключенная к нему) автоматически создает потоки серверных процедур по мере необходимости. Если приложению требуется контроль над созданием потоков, оно может его осуществить с помощью функций, описанных в разделе 15.9.

Мы также убедились, что сервер в этом случае является параллельным (`concurrent`): одновременно может выполняться несколько экземпляров процедуры сервера в виде отдельных потоков для обслуживания клиентов. Это следует также из того, что результат работы сервера выводится тремя экземплярами клиента одновременно пять секунд спустя после их одновременного запуска. Если бы сервер был последовательным, первый результат появился бы через 5 секунд после запуска, следующий — через 10, а последний — через 15.

Автоматическое управление потоками сервера: несколько процедур

В предыдущем примере процесс-сервер содержал лишь одну процедуру сервера. Вопрос, которым мы займемся теперь, звучит так: могут ли несколько процедур одного процесса использовать один и тот же пул потоков сервера? Чтобы узнать ответ, добавим к нашему серверу еще одну процедуру, а заодно перепишем наши программы заново, чтобы продемонстрировать более приличный стиль передачи аргументов и результатов между процессами.

Первый файл в этом примере называется `squareproc.h`. В нем определен один тип данных для входных аргументов функции, возводящей в квадрат, и еще один — для возвращаемых ею результатов. В этом заголовочном файле также определяется полное имя двери для данной процедуры. Его текст его приведен в листинге 15.8.

Наша новая процедура будет принимать длинное целое и возвращать квадратный корень из него (типа `double`). Мы определяем полное имя двери этой процедуры, структуры аргументов и результатов в заголовочном файле `sqrtproc.h` в листинге 15.9.

Программа-клиент приведена в листинге 15.10. Она последовательно вызывает две процедуры сервера и выводит возвращаемые ими результаты. Эта программа устроена аналогично другим клиентским программам, приведенным в этой главе.

Текст двух серверных процедур приведен в листинге 15.11. Каждая из них выводит текущий идентификатор потока и значение аргумента, делает 5-секунд-ную паузу, вычисляет результат и завершает работу.

Функция `main` сервера, текст которой приведен в листинге 15.12, открывает дескрипторы дверей и связывает каждый из них с одной из процедур сервера.

Запустим программу-клиент и подождем 10 секунд до вывода результатов (как мы и ожидали):

Посмотрев на выводимый сервером текст, мы увидим, что один и тот же поток этого процесса использовался для обработки обоих запросов клиента:

Это подтверждает наши предположения о том, что любой поток из пула сервера может использоваться при обработке запросов клиентов для любой процедуры.

Атрибут DOOR_UNREF для серверов

В разделе 15.3 мы отметили, что при вызове `door_create` для создаваемой двери можно указать атрибут `DOOR_UNREF`. В документации говорится, что если количество дескрипторов, относящихся к этой двери, уменьшается с двух до одного, осуществляется специальный вызов процедуры сервера. Особенность вызова заключается в том, что второй аргумент процедуры сервера (указатель на данные) при этом является константой `DOOR_UNREF_DATA`. Мы продемонстрируем три способа обращения к двери.

1. Дескриптор, возвращаемый `door_create`, считается первой ссылкой на эту дверь. Вообще говоря, причина, по которой специальный вызов происходит при изменении количества дескрипторов с 2 на 1, а не с 1 на 0, заключается в том, что первый дескриптор обычно не закрывается сервером до завершения работы.

2. Полное имя, связанное с дверью в файловой системе, также считается ссылкой на дверь. Ее можно удалить вызовом функции `fdetach`, или запустив программу `fdetach`, или удалив полное имя из файловой системы (функцией `unlink` или командой `rm`).

3. Дескриптор, возвращаемый клиенту функцией `open`, считается открытой ссылкой до тех пор, пока не будет закрыт либо явным вызовом `close`, либо неявно, при завершении клиента. Во всех примерах этой главы дескриптор закрывается неявно.

Первый пример показывает, что если сервер закрывает свой дескриптор после вызова `fattach`, немедленно происходит специальный вызов процедуры сервера. В листинге 15.13 приведен текст процедуры сервера и функции `main`.

7-10 Процедура сервера распознает специальный вызов и выводит сообщение об этом. Возврат из специального вызова происходит путем вызова `door_return` с двумя нулевыми указателями и нулевыми значениями размеров.

28 Теперь мы закрываем дескриптор двери после выполнения `fattach`. Этот дескриптор может быть нужен серверу только для вызовов `door_bind`, `doo_info` и `door_revoke`.

Запустив сервер, мы увидим, что немедленно произойдет специальный вызов:

Если мы проследим за значением счетчика открытых дескрипторов, мы увидим, что он становится равен 1 после возврата из `door_create` и 2 после возврата из `fattach`. Вызов `close` уменьшает количество открытых дескрипторов с двух до одного, что приводит к специальному вызову процедуры. Единственная оставшаяся ссылка при этом представляет собой имя в файловой системе, а этого клиенту достаточно, чтобы обратиться к двери. Поэтому клиент продолжает работать правильно:

Более того, дальнейших специальных вызовов серверной процедуры не происходит. Для каждой двери осуществляется только один специальный вызов.

Теперь изменим нашу программу-сервер обратно, убрав вызов `close` для дескриптора двери. Процедура сервера и функция `main` приведены в листинге 15.14.

Мы оставляем 6-секундную паузу и выводим сообщение о возврате из процедуры сервера. Запустим сервер в одном окне, а из другого проверим существование имени файла двери в файловой системе и удалим его с помощью `rm`:

После удаления имени файла происходит специальный вызов процедуры сервера:

Если мы проследим за количеством ссылок на эту дверь, то увидим следующее: одна ссылка появляется после вызова `door_create`, вторая — после `fattach`. После удаления файла с помощью `rm` количество ссылок снова уменьшается до единицы, что приводит к специальному вызову процедуры.

В последнем примере использования этого атрибута мы снова удалим имя из файловой системы. Но на этот раз мы сначала запустим три экземпляра программы-клиента. Специальный вызов процедуры произойдет только после завершения последнего клиента, потому что каждый экземпляр клиента увеличивает количество ссылок на дверь. Используем сервер из листинга 15.14

и клиент из листинга 15.2.

Сервер при этом выведет вот что:

Проследим за значением счетчика открытых ссылок для этой двери. Он становится равным 1 после вызова door_create, 2 после вызова fattach. Когда три клиента вызывают open, счетчик увеличивается с 2 до 5. После удаления имени файла счетчик уменьшается до 4. После завершения работы трех клиентов счетчик уменьшается с 4 до 1 (последовательно), и последнее его изменение с 2 до 1 приводит к специальному вызову процедуры.

Этими примерами мы показали, что хотя описание атрибута DOOR_UNREF выглядит просто («специальный вызов происходит при изменении счетчика ссылок с 2 до 1»), мы должны понимать принципы работы этого счетчика, чтобы им пользоваться.

15.8. Передача дескрипторов

Когда мы говорим о передаче открытого дескриптора от одного процесса другому, обычно подразумевается одно из двух:

- наследование всех открытых дескрипторов родительского процесса дочерним после вызова fork;
- сохранение открытых дескрипторов при вызове exec.

В первом случае процесс открывает дескриптор, вызывает fork, а затем родительский процесс закрывает дескрипторы, предоставляя дочернему возможность работать с ними. При этом открытый дескриптор передается от родительского процесса дочернему.

В современных версиях Unix возможности передачи дескрипторов существенно расширены, и теперь имеется возможность передавать открытый дескриптор от одного процесса другому вне зависимости от их родства. Двери являются одним из существующих интерфейсов для передачи дескрипторов от клиента серверу и от сервера клиенту.

ПРИМЕЧАНИЕ

Передача дескрипторов через доменные сокеты Unix была описана в разделе 14.7 [24]. В ядрах Berkeley и производных от них дескрипторы передаются именно через такие сокеты. Все подробности описаны в главе 18 [23]. В ядрах SVR4 используются другие методы передачи дескрипторов, а именно команды I_SENDFD и I_RECVFD функции ioctl. Они описаны в разделе 15.5.1 [21]. Но процесс в SVR4 может воспользоваться и механизмом доменных сокетов Unix.

Нужно правильно понимать, что именно подразумевается под передачей дескриптора. На рис. 4.7 сервер открывал файл и копировал его целиком через нижний (на рисунке) канал. Если размер файла 1 Мбайт, через канал будет передан 1 Мбайт данных. Но если сервер передает клиенту дескриптор вместо самого файла, то через канал передается только дескриптор (который содержит небольшое количество информации ядра). Клиент может использовать этот дескриптор для считывания содержимого файла. Чтение файла при этом осуществляется именно клиентом, а сервер осуществляет только открытие файла.

Нужно понимать, что сервер не может просто записать в канал числовое значение дескриптора, как в следующем фрагменте кода:

Этот подход не работает. Дескрипторы вычисляются для каждого процесса в отдельности. Предположим, что значение дескриптора файла на сервере равно 4. Даже если дескриптор с тем же значением и открыт клиентом, он почти наверняка относится к другому файлу. Единственная ситуация, в которой дескрипторы одного процесса имеют значение для другого процесса, возникает при вызове fork.

Если первый свободный дескриптор сервера имеет значение 4, вызов open вернет именно это значение. Если сервер передает дескриптор 4 клиенту, а у клиента наименьшее свободное значение дескриптора равно 7, нужно, чтобы дескриптор 7 клиента был установлен в соответствие с тем же файлом, что и дескриптор 4 сервера. Рисунки 15.4 в [21] и 18.4 в [23] иллюстрируют, что должно произойти с точки зрения ядра: два дескриптора (4 у сервера и 7 у клиента) должны указывать на один и тот же файл из таблицы ядра. Интерфейсы типа дверей и доменных сокетов Unix скрывают внутренние детали реализации, предоставляя процессам возможность легко передавать дескрипторы друг другу.

Дескрипторы передаются через дверь от клиента серверу путем присваивания полю desc_ptr структуры door_arg_t значения указателя на массив структур типа door_desc_t и помещения в поле desc_num количества этих структур. Дескрипторы передаются от сервера клиенту путем присваивания третьему аргументу door_return значения указателя на массив структур door_desc_t и помещения в четвертый аргумент количества передаваемых дескрипторов:



Рис. 15.4. Сервер файлов, передающий клиенту дескриптор

Эта структура содержит объединение (union), и первое поле структуры является тегом, идентифицирующим содержимое этого объединения. В настоящий момент определено только одно поле объединения (структура d_desc, описывающая дескриптор), и тег (d_attributes) должен иметь значение DOOR_DESCRIPTOR.

Пример

Изменим наш пример с сервером файлов таким образом, чтобы сервер открывал файл, передавал дескриптор клиенту, а клиент копировал содержимое файла в стандартный поток вывода. На рис. 15.4 приведена схема приложения. В листинге 15.15 приведен текст программы клиента.

9-15 Имя файла, связанного с дверью, принимается в качестве аргумента командной строки. Имя файла, который должен быть открыт и выведен, считывается из стандартного потока ввода, а завершающий символ перевода строки удаляется.

16-22 Подготавливается структура door_arg_t. К размеру имени файла мы добавляем единицу, чтобы сервер мог дополнить его завершающим нулем.

23-31 Мы вызываем процедуру сервера и проверяем результат. Должен возвращаться только один дескриптор и никаких данных. Вскоре мы увидим, что сервер возвращает данные (сообщение об ошибке) только в том случае, если он не может открыть файл. В этом случае функция err_quit выводит сообщение об ошибке.

32-34 Дескриптор извлекается из структуры door_desc_t, и файл копируется в стандартный поток вывода.

В листинге 15.16 приведен текст процедуры сервера. Функция main по сравнению с листингом 15.3 не изменилась.

9-14 Мы завершаем полное имя файла клиента нулем и делаем попытку открыть этот файл вызовом open. Если возникает ошибка, сообщение о ней возвращается клиенту.

15-20 Если файл был успешно открыт, клиенту возвращается только его дескриптор.

Запустим сервер и укажем ему имя двери /tmp/fd1, а затем запустим клиент:

В первых двух случаях мы указываем имя файла, приводящее к возврату сообщения об ошибке. В третий раз сервер передает клиенту дескриптор файла из двух строк, который благополучно выводится.

ПРИМЕЧАНИЕ

Существует проблема, связанная с передачей дескриптора через дверь. Чтобы она проявилась в нашем примере, достаточно добавить вызов printf к процедуре сервера сразу после успешного вызова open. Вы увидите, что значение дескриптора каждый раз увеличивается на единицу. Проблема в том, что сервер не закрывает дескрипторы после передачи их клиенту. Сделать это, вообще говоря, нелегко. Логично было бы выполнять закрытие дескриптора после возврата из door_return, после успешной отправки дескриптора клиенту, но возврата из door_return не происходит! Если бы мы использовали sendmsg для передачи дескриптора через доменный сокет Unix или ioctl для передачи дескриптора через канал в SVR4, мы могли бы закрыть его после возврата из sendmsg или ioctl. Однако с дверьми все по-другому, поскольку возврата из функции door_return не происходит. Единственный способ обойти проблему заключается в том, что процедура сервера должна запоминать все открытые дескрипторы и закрывать их некоторое время спустя, что несколько запутывает код.

Эта проблема должна быть исправлена в Solaris 2.7 добавлением атрибута DOOR RELEASE. Отправитель устанавливает поле d_attributes равным DOOR_DESCRIPTOR | DOOR_RELEASE, что говорит системе о необходимости закрывать дескриптор после передачи его клиенту.

15.9. Функция door_server_create

В листинге 15.6 мы показали, что библиотека дверей автоматически создает новые потоки для обслуживания запросов клиентов по мере их поступления. Они создаются библиотекой как неприсоединенные потоки (*detached threads*) с размером стека потока по умолчанию, с отключенной возможностью отмены потока (*thread cancellation*) и с маской сигналов и классом планирования (*scheduling class*), унаследованными от потока, вызвавшего `door_create`. Если мы хотим изменить какой-либо из этих параметров или хотим самостоятельно работать с пулом потоков сервера, можно воспользоваться функцией `door_server_create` и указать нашу собственную процедуру создания сервера:

Как и при объявлении `door_create` в разделе 15.3, мы используем оператор `typedef` для упрощения прототипа библиотечной функции. Наш новый тип данных определяет процедуру создания сервера как принимающую один аргумент (указатель на структуру типа `door_info_t`) и ничего не возвращающую (`void`). При вызове `door_server_create` аргументом является указатель на нашу процедуру создания сервера, а возвращается указатель на предыдущую процедуру создания сервера.

Наша процедура создания сервера вызывается при возникновении необходимости создания нового потока для обслуживания запроса клиента. Информация о том, какой из процедур сервера требуется новый поток, передается в структуре `door_info_t`, адрес которой принимается процедурой создания сервера. Поле `di_proc` содержит адрес процедуры сервера, а поле `di_data` содержит указатель на аргументы, передаваемые процедуре сервера при вызове.

Проще всего изучить происходящее на примере. Программа-клиент не претерпевает никаких изменений по сравнению с листингом 15.1. В программу-сервер добавляются две новые функции помимо процедуры сервера и функции `main`. На рис. 15.5 приведена схема сервера с четырьмя функциями и последовательностью их регистрации и вызова.



Рис. 15.5. Четыре функции в процессе-сервере

В листинге 15.17 приведен текст функции `main` сервера.

По сравнению с листингом 15.2 было внесено четыре изменения:

1. Убрано объявление дескриптора двери `fd` (теперь это глобальная переменная, описанная в листинге 15.18).
2. Вызов `door_create` защищен взаимным исключением (также описанным в листинге 15.18).
3. Вызов `door_server_create` делается перед созданием двери, при этом указывается процедура создания сервера (`my_thread`, которая, будет показана позже).
4. В вызове `door_create` последний аргумент (атрибуты) имеет значение `DOOR_PRIVATE` вместо 0. Это говорит библиотеке о том, что данная дверь будет иметь собственный пул потоков, называемый частным пулом сервера.

Задание процедуры создания сервера с помощью `door_server_create` и выделение частного пула сервера с помощью `DOOR_PRIVATE` осуществляются независимо друг от друга. Возможны четыре ситуации:

1. По умолчанию частный пул сервера и процедура создания сервера отсутствуют. Система создает потоки по мере необходимости и они переходят в пул потоков процесса.
2. Указан флаг `DOOR_PRIVATE`, но процедура создания сервера отсутствует. Система создает потоки по мере необходимости и они отходят в пул потоков процесса, если относятся к тем дверям, для которых флаг `DOOR_PRIVATE` не был указан, либо в пул данной двери, если она была создана с флагом `DOOR_PRIVATE`.
3. Отсутствует частный пул сервера, но указана процедура создания сервера. Процедура создания вызывается при необходимости создания нового потока, который затем переходит в пул потоков процесса.
4. Указан флаг `DOOR_PRIVATE` и процедура создания сервера. Процедура создания сервера вызывается каждый раз при необходимости создания потока. После создания поток должен вызвать `door_bind` для отнесения его к нужному частному пулу сервера, иначе он будет добавлен к пулу

потоков процесса.

В листинге 15.18 приведен текст двух новых функций: `my_create` (процедура создания сервера) и `my_thread` (функция, выполняемая каждым потоком, который создается `my_create`).

30-41 Каждый раз при вызове `my_create` создается новый поток. Перед вызовом `pthread_create` атрибуты потока инициализируются, область потока устанавливается равной `PTHREAD_SCOPE_SYSTEM` и поток определяется как неприсоединенный (`detached`). Созданный поток вызывает функцию `my_thread`. Аргументом этой функции является указатель на структуру типа `door_info_t`. Если у нас имеется сервер с несколькими дверьми и мы указываем процедуру создания сервера, эта процедура создания сервера будет вызываться при необходимости создания потока для любой из дверей. Единственный способ, которым эта процедура может определить тип сервера, соответствующий нужной двери, заключается в изучении указателя `di_proc` в структуре типа `door_info_t`.

ПРИМЕЧАНИЕ

Установка области выполнения `PTHREAD_SCOPE_SYSTEM` означает, что поток будет конкурировать в распределении ресурсов процессора с другими потоками системы. Альтернативой является указание `PTHREAD_SCOPE_PROCESS`; при этом поток будет конкурировать только с другими потоками данного процесса. Последнее не будет работать с дверьми, поскольку библиотека дверей требует, чтобы тот процесс ядра, который привел к вызову данного потока, выполнял и `door_return`. Поток с `PTHREAD_SCOPE_PROCESS` может сменить поток ядра во время выполнения процедуры сервера.

Причина, по которой поток должен создаваться как неприсоединенный, заключается в том, что нужно предотвратить сохранение в системе информации о потоке после его завершения, потому что отсутствует поток, вызывающий `pthread_join`.

15-20 При создании потока запускается функция `my_thread`, указанная в вызове `pthread_create`. Аргументом является указатель на структуру типа `door_info_t`, передаваемый `my_create`. В данном примере есть только одна процедура сервера — `servproc`, и мы просто проверяем, что аргумент указывает на эту процедуру.

21-22 Процедура создания сервера вызывается в первый раз при вызове `door_create` для создания первого потока сервера. Этот вызов осуществляется из библиотеки дверей до завершения работы `door_create`. Однако переменная `fd` не примет значения дескриптора двери до тех пор, пока не произойдет возврата из функции `door_create` (проблема курицы и яйца). Поскольку мы знаем, что `my_thread` выполняется отдельно от основного потока, решение состоит в том, чтобы использовать взаимное исключение `fdlock` следующим образом: основной поток блокирует взаимное исключение перед вызовом `door_create` и разблокирует после возврата из `door_create` (когда дескриптору `fd` уже присвоено некоторое значение). Функция `my_thread` делает попытку заблокировать взаимное исключение (ее выполнение приостанавливается до тех пор, пока основной поток не разблокирует это взаимное исключение), а затем разблокирует его. Мы могли бы добавить условную переменную и передавать по ней уведомление, но здесь это не нужно, поскольку мы заранее знаем, в каком порядке будут происходить вызовы.

23 При создании нового потока вызовом `pthread_create` его отмена по умолчанию разрешена. Если отмена потока разрешена и клиент прерывает вызов `door_call` в процессе его выполнения (что мы продемонстрируем в листинге 15.26), вызываются обработчики отмены потока, после чего он завершается. Если отмена потока отключена (как это делаем мы) и клиент прерывает работу в вызове `door_call`, процедура сервера спокойно завершает работу (поток не завершается), а результаты `door_return` просто сбрасываются. Поскольку серверный поток завершается, если происходит отмена потока, и поскольку процедура сервера может в этот момент выполнять какие-то действия (возможно, с заблокированными семафорами или блокировками), библиотека дверей на всякий случай отключает отмену всех создаваемых ею потоков. Если нам нужно, чтобы процедура сервера отменялась при досрочном завершении работы клиента, для этого потока следует включить возможность отмены и приготовиться обработать такую ситуацию.

ПРИМЕЧАНИЕ

Обратите внимание, что область выполнения `PTHREAD_SCOPE_SYSTEM` и неприсоединенность потока указываются как атрибуты при создании потока. А отмена потока может быть отключена только в процессе выполнения потока. Таким образом, хотя мы и отключаем отмену потока, он сам может ее включить и выключить тогда, когда потребуется.

24 Вызов `door_bind` позволяет добавить поток к пулу, связанному с дверью, дескриптор которой

передается `door_bind` в качестве аргумента. Поскольку для этого нам нужно знать дескриптор двери, в этой версии сервера он является глобальной переменной.

25 Мы делаем поток доступным клиенту вызовом `door_return` с двумя нулевыми указателями и нулевыми значениями длин буферов в качестве аргументов.

Процедура сервера приведена в листинге 15.19. Она идентична программе из листинга 15.6.

Чтобы продемонстрировать работу программы, запустим сервер:

После запуска сервера и вызова `door_create` процедура создания сервера запускается в первый раз, хотя клиент мы еще не запустили. При этом создается первый поток, ожидающий запроса от первого клиента. Затем мы запускаем клиент три раза подряд:

Посмотрим, что при этом выводит сервер. При поступлении первого запроса клиента создается новый поток (с идентификатором потока 5), а поток с номером 4 обслуживает все запросы клиентов. Библиотека дверей всегда держит один лишний поток наготове:

Запустим теперь три экземпляра клиента одновременно в фоновом режиме:

Посмотрев на вывод сервера, мы увидим, что было создано два новых потока (с идентификаторами 6 и 7) и потоки 4, 5 и 6 обслужили три запроса от клиентов:

15.10. Функции door_bind, door_unbind и door_revoke

Рассмотрим еще три функции, дополняющие интерфейс дверей:

Функция `door_bind` впервые появилась в листинге 15.18. Она связывает вызвавший ее поток с частным пулом сервера, относящимся к двери с дескриптором `fd`. Если вызвавший поток уже подключен к какой-либо другой двери, производится его неявное отключение.

Функция `door_unbind` осуществляет явное отключение потока от текущего пула, к которому он подключен.

Функция `door_revoke` отключает доступ к двери с дескриптором `fd`. Дескриптор двери может быть отменен только процессом, создавшим эту дверь. Все вызовы через эту дверь, находящиеся в процессе выполнения в момент вызова этой функции, будут благополучно завершены.

В наших примерах до настоящего момента предполагалось, что в процессе работы клиента и сервера не возникает непредусмотренных ситуаций. Посмотрим, что произойдет, если у клиента или сервера возникнут ошибки. В случае если клиент и сервер являются частями одного процесса (локальный вызов процедуры на рис. 15.1), клиенту не нужно беспокоиться о возникновении ошибок на сервере, и наоборот. Однако если клиент и сервер находятся в различных процессах, нужно учесть возможность досрочного завершения одного из них и предусмотреть способ уведомления второго об этом событии. Об этом нужно заботиться вне зависимости от того, находятся ли клиент и сервер на одном узле или нет.

Досрочное завершение сервера

Если клиент блокируется в вызове `door_call`, ожидая получения результатов, ему нужно каким-то образом получить уведомление о завершении потока сервера по какой-либо причине. Посмотрим, что происходит в этом случае, прервав работу сервера вызовом `pthread_exit`. Это приведет к завершению потока сервера (а не всего процесса). В листинге 15.20 приведен текст процедуры сервера.

Оставшаяся часть сервера не претерпевает изменений по сравнению с листингом 15.2, а программу-клиент мы берем из листинга 15.1.

Запустив клиент, мы увидим, что вызов `door_call` возвращает ошибку `EINTR`, если процедура сервера завершается досрочно:

Непрерываемость системного вызова door_call

Документация на door_call предупреждает, что эта функция не предполагает возможности перезапуска (библиотечная функция door_call делает системный вызов с тем же именем). Мы можем убедиться в этом, изменив процедуру сервера таким образом, чтобы она делала паузу в 6 секунд перед возвращением, что показано в листинге 15.21.

Изменим теперь клиент из листинга 15.2: установим обработчик сигнала SIGCHLD, добавив порождение процесса и завершение порожденного процесса через 2 секунды. Таким образом, через 2 секунды после вызова door_call дочерний процесс завершит работу, а родительский перехватит сигнал SIGCHLD и произойдет возврат из обработчика сигнала, прерывающий системный вызов door_call. Текст программы-клиента показан в листинге 15.22.

Клиенту будет возвращена та же ошибка, что и при досрочном завершении сервера — EINTR:

Поэтому нужно блокировать все сигналы, которые могут прервать вызов door_call.

Идемпотентные и неидемпотентные процедуры

А что произойдет, если мы перехватим сигнал EINTR и вызовем процедуру сервера еще раз, поскольку мы знаем, что эта ошибка возникла из-за нашего собственного прерывания системного вызова перехваченным сигналом (SIGCHLD)? Это может привести к некоторым проблемам, как мы покажем ниже.

Изменим сервер так, чтобы он выводил идентификатор вызванного потока, делал паузу в 6 секунд и выводил идентификатор потока по завершении его. В листинге 15.23 приведен текст новой процедуры сервера.

В листинге 15.24 приведен текст программы-клиента.

2-8 Объявляем глобальную переменную `caught_sigchld`, устанавливая ее в единицу при перехвате сигнала SIGCHLD.

31-42 Вызываем `door_call` в цикле, пока он не завершится успешно.

Глядя на выводимые клиентом результаты, мы можем подумать, что все в порядке:

Функция `door_call` вызывается в первый раз, обработчик сигнала срабатывает через 2 секунды после этого и переменной `caught_sigchld` присваивается значение 1. `door_call` при этом возвращает ошибку EINTR и мы вызываем `door_call` еще раз. Во второй раз процедура завершается успешно.

Посмотрев на выводимый сервером текст, мы увидим, что процедура сервера была вызвана дважды:

Когда клиент второй раз вызывает `door_call`, это приводит к запуску нового потока, вызывающего процедуру сервера еще раз. Если процедура сервера идемпотентна, проблем в такой ситуации не возникнет. Однако если она неидемпотентна, это может привести к ошибкам.

Термин «идемпотентность» по отношению к процедуре подразумевает, что процедура может быть вызвана произвольное число раз без возникновения ошибок. Наша процедура сервера, вычисляющая квадрат целого числа, идемпотентна: мы получаем правильный результат вне зависимости от того, сколько раз мы ее вызовем. Другим примером является процедура, возвращающая дату и время. Хотя эта процедура и будет возвращать разную информацию при новых вызовах (поскольку дата и время меняются), это не вызовет проблем. Классическим примером неидемпотентной процедуры является процедура уменьшения банковского счета на некоторую величину. Конечный результат будет неверным, если ее вызвать дважды.

Досрочное завершение клиента

Посмотрим, каким образом процедура сервера получает уведомление о досрочном завершении клиента. Программа-клиент приведена в листинге 15.25.

20 Единственное изменение заключается в добавлении вызова `alarm(3)` перед `door_call`. Эта функция приводит к отправке сигнала `SIGALRM` через три секунды после вызова, но, поскольку мы его не перехватываем, это приводит к завершению процесса. Поэтому клиент завершится до возврата из `door_call`, потому что в процедуре сервера вставлена шестисекундная пауза.

В листинге 15.26 приведен текст процедуры сервера и обработчик отмены потока.

Вспомните, что мы говорили об отмене выполнения потока в разделе 8.5 и в связи с листингом 15.18. Когда система обнаруживает завершение клиента в процессе выполнения серверной процедуры, потоку, обрабатывающему запрос этого клиента, отправляется запрос на отмену:

- если поток отключил возможность отмены, ничего не происходит и поток выполняется до завершения (вызов `door_return`), а результаты сбрасываются;
- если возможность отмены включена, вызываются обработчики отмены потока, а затем он завершает работу.

В тексте процедуры сервера мы сначала вызвали `pthread_setcancelstate` для включения возможности отмены потока, потому что по умолчанию при создании новых потоков библиотекой возможность их отмены отключается. Эта функция сохраняет текущее состояние потока в переменной `oldstate`, и мы восстанавливаем его в конце функции. Затем мы вызываем `pthread_cleanup_push` для регистрации нашего обработчика отмены `servproc_cleanup`. Эта функция только выводит идентификатор отмененного потока, но вообще она может выполнять все необходимое для корректного завершения процедуры сервера (разблокировать исключения и т. п.). После возвращения из обработчика поток завершается.

В тексте процедуры сервера мы добавляем 6-секундную паузу, чтобы клиент мог успешно завершить работу в вызове `door_call`.

Запустив клиент дважды, мы увидим сообщение интерпретатора Alarm clock при завершении процесса сигналом `SIGALRM`:

Посмотрим, что при этом выводит сервер. Каждый раз при досрочном завершении клиента поток процедуры сервера действительно отменяется и вызывается обработчик отмены потока:

Цель, с которой мы вызываем программу-клиент дважды, — показать, что после завершения потока с идентификатором 4 библиотека создает новый поток (с идентификатором 5) для обработки второго запроса клиента.

15.12. Резюме

Интерфейс дверей позволяет вызывать процедуры в других процессах на том же узле. В следующей главе мы обсудим возможность удаленного вызова процедур в процессах на других узлах.

Основные функции этого интерфейса просты в работе и использовании. Сервер вызывает `door_create` для создания двери и связывания ее с процедурой сервера, а затем вызывает `fattach` для сопоставления этой двери и имени файла в файловой системе. Клиент вызывает `open` для этого имени файла и затем может вызвать `door_call` для вызова процедуры сервера. Возврат из процедуры сервера осуществляется вызовом `door_return`.

Обычно разрешения для двери проверяются только один раз — при ее открытии вызовом `open`. Проверяются идентификаторы пользователя и группы клиента (и полного имени файла). Одной из полезных функций дверей (по сравнению с другими средствами IPC) является возможность получения информации о клиенте в процессе работы (его действующего и реального идентификаторов). Это может использоваться сервером для принятия решения о предоставлении услуг данному клиенту.

Двери предоставляют возможность передачи дескрипторов от клиента серверу и обратно. Это достаточно мощное средство, поскольку дескрипторы в Unix дают возможность обращаться ко множеству объектов (файлам, сокетам или XTI, дверям).

При вызове процедур в другом процессе следует учесть возможность досрочного завершения клиента или сервера. Клиент получает уведомление о досрочном завершении сервера с помощью сообщения об ошибке `EINTR`. Сервер получает уведомление о досрочном завершении клиента в процессе обработки процедуры посредством запроса на отмену выполнения потока данной процедуры. Сервер может обработать этот запрос или проигнорировать его.

Упражнения

1. Сколько байтов информации передается при вызове `door_call` от клиента серверу?
2. Есть ли необходимость вызывать `fstat` для проверки типа дескриптора в листинге 15.3? Уберите этот вызов и посмотрите, что произойдет.
3. В документации Solaris 2.6 для вызова `sleep()` говорится, что «выполнение текущего процесса приостанавливается». Почему при этом библиотека дверей имеет возможность создать новые потоки в листинге 15.6?
4. В разделе 15.3 мы отмечали, что для создаваемых вызовом `door_create` дверей автоматически устанавливается бит `FD_CLOEXEC`. Однако мы можем вызвать `fcntl` после возврата из `door_create` и сбросить этот бит. Что произойдет, если мы сделаем это, вызовем `exec`, а затем обратимся к процедуре сервера из клиента?
5. В листингах 15.23 и 15.24 добавьте вывод текущего времени в вызовах `printf` сервера и клиента. Запустите клиент и сервер. Почему первый экземпляр процедуры сервера возвращается через две секунды после запуска?
6. Удалите блокировку, защищающую дескриптор `fd` в листингах 15.17 и 15.18, и убедитесь, что программа больше не работает. Какая при этом возникает ошибка?
7. Если мы хотим лишь испытать возможность отмены потока с процедурой сервера, нужно ли нам устанавливать процедуру создания сервера?
8. Проверьте, что вызов `door_revoke` дает возможность завершиться работающим с данной процедурой потокам. Выясните, что происходит при вызове `door_call` после аннулирования процедуры.
9. В нашем решении предыдущего упражнения и в листинге 15.17 мы говорим, что дескриптор двери должен быть глобальным, если он нужен процедуре сервера или процедуре создания сервера. Это утверждение, вообще говоря, неверно. Перепишите решение предыдущего упражнения, сохранив `fd` в качестве автоматической переменной функции `main`.
10. В программе листинга 15.18 мы вызывали `pthread_attr_init` и `pthread_attr_destroy` каждый раз, когда создавался поток. Является ли такое решение оптимальным?

Когда мы разрабатываем приложение, мы встаем перед выбором:

1. Написать одну большую монолитную программу, которая будет делать все.
2. Разделить приложение на несколько процессов, взаимодействующих друг с другом.

Если мы выбираем второй вариант, перед нами опять встает вопрос:

2.1. предполагать ли, что все процессы выполняются на одном узле, что допускает использование средств IPC для взаимодействия между ними, либо

2.2. допускать возможность запуска части процессов на других узлах, что требует какой-либо формы взаимодействия по сети.

Взгляните на рис. 15.1. Верхняя ситуация соответствует варианту 1, средняя — варианту 2.1, а нижняя — варианту 2.2. Большая часть этой книги посвящена обсуждению случая 2.1, то есть взаимодействию процессов в пределах одного узла с помощью таких средств IPC, как передача сообщений, разделяемая память и, возможно, некоторых форм синхронизации. Взаимодействие между потоками одного процесса или разных процессов является частным случаем этого сценария.

Когда мы выдвигаем требование поддержки возможности сетевого взаимодействия между составляющими приложения, для разработки чаще всего используется явное сетевое программирование. При этом в программе используются вызовы интерфейсов сокетов или ХTI, о чем рассказывается в [24]. При использовании интерфейса сокетов клиенты вызывают функции socket, connect, read и write, а серверы вызывают socket, bind, listen, accept, read и write. Большая часть знакомых нам приложений (браузеры, серверы Web, клиенты и серверы Telnet) написаны именно так.

Альтернативным способом разработки распределенных приложений является неявное сетевое программирование. Оно осуществляется посредством удаленного вызова процедур (remote procedure call — RPC). Приложение кодируется с использованием тех же вызовов, что и обычное несетевое, но клиент и сервер могут находиться на разных узлах. Сервером называется процесс, предоставляющий другому возможность запускать одну из составляющих его процедур, а клиентом — процесс,зывающий процедуру сервера. То, что клиент и сервер находятся на разных узлах и для связи между ними используется сетевое взаимодействие, большей частью остается скрыто от программиста. Одним из критериев оценки качества пакета RPC является именно скрытость лежащего в основе интерфейса сети.

Пример

В качестве примера использования RPC перепишем листинги 15.1 и 15.2 для использования Sun RPC вместо дверей. Клиент вызывает процедуру сервера с аргументом типа long, а возвращаемое значение представляет собой квадрат аргумента. В листинге 16.1 приведен текст первого файла, square.x.

Файлы с расширением .x называются файлами спецификации RPC. Они определяют процедуры сервера, их аргументы и возвращаемые значения.

1-6 Мы определяем две структуры — одну для аргументов (одно поле типа long) и одну для результатов (тоже одно поле типа long).

7-11 Мы определяем программу RPC с именем SQUARE_PROG, состоящую из одной версии (SQUARE_VERS), а эта версия представляет собой единственную процедуру сервера с именем SQUAREPROC. Аргументом этой процедуры является структура square_in, а возвращаемым значением — структура square_out. Мы также присваиваем этой процедуре номер 1, как и версии, а программе мы присваиваем 32-разрядный шестнадцатеричный номер. (О номерах программ более подробно говорится в табл. 16.1.)

Компилировать спецификацию нужно программой grcsen, входящей в пакет Sun RPC.

Теперь напишем функцию main клиента, который будет осуществлять удаленный вызов процедуры. Текст программы приведен в листинге 16.2.

2 Мы подключаем заголовочный файл square.h, создаваемый функцией grcsen.

6 Мы объявляем дескриптор клиента (client handle) с именем cl. Дескрипторы клиентов выглядят как обычные указатели на тип FILE (поэтому слово CLIENT пишется заглавными буквами).

11 Мы вызываем функцию clnt_create, создающую клиент RPC:

Как и с обычными указателями на тип FILE, нам безразлично, на что указывает дескриптор клиента. Скорее всего, это некоторая информационная структура, хранящаяся в ядре. Функция clnt_create создает такую структуру и возвращает нам указатель на нее, а мы передаем его библиотеке RPC времени выполнения каждый раз при удаленном вызове процедуры.

Первым аргументом clnt_create должно быть имя или IP-адрес узла, на котором выполняется сервер. Вторым аргументом будет имя программы, третьим — номер версии. Оба эти значения берутся из спецификации (square.x, листинг 16.1). Последний аргумент позволяет указать протокол, обычно TCP или UDP.

12-15 Мы вызываем процедуру, причем первый аргумент указывает на входную структуру (&in), а второй содержит дескриптор клиента. В большинстве стандартных функций ввода-вывода дескриптор файла является последним аргументом. Точно так же и в вызовах RPC дескриптор клиента передается последним. Возвращаемое значение представляет собой указатель на структуру, в которой хранится результат работы сервера. Место под входную структуру выделяет программист, а под возвращаемую — пакет RPC.

В файле спецификации square.x мы назвали процедуру SQUAREPROC, но из клиента мы вызываем squareproc_1. Существует соглашение о преобразовании имени из файла спецификации к нижнему регистру и добавлении номера версии через символ подчеркивания.

Со стороны сервера от нас требуется только написать процедуру. Функция main автоматически создается программой grcsen. Текст процедуры приведен в листинге 16.3.

3-4 Прежде всего мы замечаем, что к имени процедуры добавился суффикс _svc. Это дает возможность использовать два прототипа функций ANSI C в файле square.x, один из которых определяет функцию, вызываемую клиентом в листинге 16.2 (она принимает дескриптор клиента), а второй — реальную функцию сервера (которая принимает другие аргументы).

При вызове процедуры сервера первый аргумент является указателем на входную структуру, а второй — на структуру, передаваемую библиотекой RPC времени выполнения, которая содержит

информацию о данном вызове (в этом примере игнорируется для простоты).

6-8 Программа считывает входной аргумент и возводит его в квадрат. Результат сохраняется в структуре, адрес которой возвращается процедурой сервера. Поскольку мы возвращаем адрес переменной, эта переменная не может быть автоматической. Мы объявляем ее как статическую (static).

ПРИМЕЧАНИЕ

Внимательные читатели заметят, что это лишает нашу функцию возможности использования в защищенном поточном программировании. Мы обсудим это в разделе 16.2, где приведем пример защищенной функции.

Откомпилируем клиент в системе Solaris, а сервер — в BSD/OS, запустим сервер, а затем клиент:

В первом случае мы указываем имя узла сервера, а во втором — его IP-адрес. Этим мы демонстрируем возможность использования как имен, так и IP-адресов для задания узла в функции `clnt_create`.

Теперь продемонстрируем некоторые ошибки, возникающие при работе `clnt_create`, если, например, не существует узел или на нем не запущена программа-сервер:

Мы написали клиентскую и серверную части программы и продемонстрировали их использование вообще без явного сетевого программирования. Клиент просто вызывает две функции, а сервер вообще состоит из одной функции. Все тонкости использования XTI в Solaris, сокетов в BSD/OS и сетевого ввода-вывода обрабатываются библиотекой RPC времени выполнения. В этом и состоит предназначение RPC — предоставлять возможность создания распределенных приложений без знания сетевого программирования.

Другая немаловажная деталь данного примера заключается в том, что в системах Sparc под Solaris и Intel x86 под управлением BSD/OS используется разный порядок байтов. В Sparc используется порядок big endian («тупоконечный»), а в Intel — little endian («остроконечный») (что мы показали в разделе 3.4 [24]). Отличия в порядке байтов также обрабатываются библиотекой RPC времени выполнения автоматически с использованием стандарта XDR (внешнее представление данных), который мы обсудим в разделе 16.8.

Создание программы-клиента и программы-сервера в данном случае требует больше операций, чем для любой другой программы этой книги. Выполняемый файл клиента получается следующим образом:

Параметр `-C` говорит `grcsen` о необходимости создания прототипов функций ANSI C в заголовочном файле `square.h`. Программа `grcsen` также создает заглушку клиента (client stub) в файле с именем `square_clnt.c` и файл с именем `square_xdr.c`, который осуществляет преобразование данных в соответствии со стандартом XDR. Наша библиотека (содержащая функции, используемые в этой книге) называется `libunprpc.a`, а параметр `-lns1` подключает системную библиотеку сетевых функций в Solaris (включая библиотеки RPC и XDR времени выполнения).

Аналогичные команды используются для создания сервера, хотя `grcsen` уже не нужно запускать снова. Файл `square_svc.c` содержит функцию `main` сервера, и файл `square_xdr.o`, обсуждавшийся выше, также требуется для работы сервера:

При этом создаются клиент и сервер, выполняемые в системе Solaris.

Если клиент и сервер должны быть построены для разных систем (как в предыдущем примере, где клиент выполнялся в Solaris, а сервер — в BSD/OS), могут потребоваться дополнительные действия. Например, некоторые файлы должны быть либо общими (через NFS), либо находиться в обеих системах, а файлы, используемые клиентом и сервером (например, `square_xdr.o`), должны компилироваться в каждой системе в отдельности.



Рис. 16.1. Этапы создания приложения клиент-сервер с использованием RPC

На рис. 16.1 приведена схема создания приложения типа клиент-сервер. Три затемненных прямоугольника соответствуют файлам, которые мы должны написать. Штриховые линии показывают файлы, подключаемые через заголовочный файл `square.h`.

На рис. 16.2 изображена схема происходящего при удаленном вызове процедуры. Действия выполняются в следующем порядке:

1. Запускается сервер, который регистрируется в программе, управляющей портами на узле-сервере. Затем запускается клиент и вызывает `clnt_create`. Эта функция связывается с управляющей портами программой сервера и находит нужный порт. Функция `clnt_create` также устанавливает соединение с сервером по протоколу TCP (поскольку мы указали TCP в качестве используемого протокола в листинге 16.2). Мы не показываем эти шаги на рисунке и откладываем детальное обсуждение до раздела 16.3.

2. Клиент вызывает локальную процедуру, называемую заглушкой клиента. В листинге 16.2 эта процедура называлась `squaregroc_1`, а файл, содержащий ее, создавался грсген автоматически и получал название `square_clnt.c`. С точки зрения клиента именно эта функция является сервером, к которому он обращается. Целью создания заглушки является упаковка аргументов для удаленного вызова процедуры, помещение их в стандартный формат и создание одного или нескольких сетевых сообщений. Упаковка аргументов клиента в сетевое сообщение называется сортировкой (marshaling). Клиент и заглушка обычно вызывают библиотеки функций RPC (`clnt_create` в нашем примере). При использовании редактора связей в Solaris эти функции загружаются из библиотеки `-lnsl`, тогда как в BSD/OS они входят в стандартную библиотеку языка C.

3. Сетевые сообщения отсылаются на удаленную систему заглушкой клиента. Обычно это требует локального системного вызова (например, `write` или `sendto`).

4. Сетевые сообщения передаются на удаленную систему. Для этого обычно используются сетевые протоколы TCP и UDP.

5. Заглушка сервера ожидает запросов от клиента на стороне сервера. Она рассортирует аргументы из сетевых сообщений.

6. Заглушка сервера осуществляет локальный вызов процедуры для запуска настоящей функции сервера (процедуры `squaregroc_1_svc` в листинге 16.3), передавая ей аргументы, полученные в сетевых сообщениях от клиента.

7. После завершения процедуры сервера управление возвращается заглушки сервера, которой передаются все необходимые значения.

8. Заглушка сервера преобразовывает возвращаемые значения к нужному формату и рассортирует их в сетевые сообщения для отправки обратно клиенту.

9. Сообщения передаются по сети обратно клиенту.

10. Заглушка клиента считывает сообщения из локального ядра (вызовом `read` или `recvfrom`).

11. После возможного преобразования возвращаемых значений заглушки клиента передает их функции клиента. Этот этап воспринимается клиентом как завершение работы процедуры.



Рис. 16.2. Действия, происходящие при удаленном вызове процедуры

История

Наверное, одна из самых старых книг по RPC — это [26]. Как пишет [4], Уайт (White) затем перешел в Xerox и там создал несколько систем RPC. Одна из них была выпущена в качестве отдельного продукта в 1981 году под именем Courier. Классической книгой по RPC является [2]. В ней описаны средства RPC проекта Cedar, работавшего на однопользовательских рабочих станциях Dorado в фирме Xerox в начале 80-х. Xerox реализовал RPC на рабочих станциях еще до того, как большинство людей узнало о том, что рабочие станции существуют! Реализация Courier для Unix распространялась много лет с версиями BSD 4.x, но в настоящий момент эта система RPC представляет только исторический интерес.

Sun выпустила первую версию пакета RPC в 1985. Она была разработана Бобом Лайоном (Bob Lyon), ушедшем в Sun из фирмы Xerox в 1983. Официально она называлась ONC/RPC: Open Network Computing Remote Procedure Call (удаленный вызов процедур в открытых вычислительных сетях), но обычно ее называют просто Sun RPC. Технически она аналогична Courier. Первые версии Sun RPC были написаны с использованием интерфейса сокетов и работали с протоколами TCP и UDP. Общедоступный исходный код вышел под названием RPCSRC. В начале 90-х он был переписан под интерфейс транспортного уровня TLI (предшественник XTI), который описан в четвертой части [24]. Теперь этот код работает со всеми протоколами, поддерживаемыми ядром. Общедоступный исходный код обеих версий можно найти по адресу <ftp://playground.sun.com/pub/grpc>, причем версия, использующая сокеты, называется gRPCSRC, а версия, использующая TLI, называется tigrpcsrc (название TI-RPC образовано от Transport Independent — транспортно-независимый удаленный вызов процедур).

Стандарт RFC 1831 [18] содержит обзор средств Sun RPC и описывает формат сообщений RPC, передаваемых по сети. Стандарт RFC 1832 [19] содержит описание XDR — и поддерживаемых типов данных, и формата их передачи «по проводам». Стандарт RFC 1833 [20] описывает протоколы привязки: RPCBIND и его предшественника, программу отображения портов (port mapper).

Одним из наиболее распространенных приложений, использующих Sun RPC, является сетевая файловая система Sun (NFS). Обычно она не создается стандартными средствами RPC, описанными в этой главе (grcsen и библиотека времени выполнения). Вместо этого большинство подпрограмм библиотеки оптимизируются вручную и добавляются в ядро с целью ускорить их работу. Тем не менее большинство систем, поддерживающих NFS, также поддерживают и Sun RPC.

В середине 80-х Apollo соперничала с Sun за рынок рабочих станций и создала свой собственный пакет RPC, призванный вытеснить Sun RPC. Этот новый пакет назывался NCA (Network Computing Architecture — архитектура сетевых вычислений). Протоколом RPC являлся протокол NCA/RPC, а аналогом XDR была схема NDR (Network Data Representation — сетевое представление данных). Интерфейсы между клиентами и серверами определялись с помощью языка NIDL (Network Interface Definition Language — язык определений сетевого интерфейса) аналогично нашему файлу .x из листинга 16.1. Библиотека времени выполнения называлась NCK (Network Computing Kernel).

Фирма Apollo была куплена Hewlett-Packard в 1989, и NCA переросла в OSF's DCE (Distributed Computing Environment — среда распределенных вычислений), основной частью которой является RPC. Более подробно о DCE можно узнать по адресу <http://www.camp.opengroup.org/tech/dce>. Реализация пакета DCE RPC свободно доступна на <ftp://gatekeeper.dec.com/pub/DEC/DCE>. Этот каталог также содержит описание внутреннего устройства пакета DCE RPC на 171 странице. Существует много версий DCE для разных платформ.

ПРИМЕЧАНИЕ

Sun RPC распространен шире, чем DCE RPC, возможно, благодаря свободно доступной реализации и включению в основную поставку большинства систем Unix. DCE RPC обычно поставляется в качестве дополнения (за дополнительную сумму). Переписывание свободно доступного кода под другие платформы пока еще не пошло полным ходом, хотя уже готовится версия для Linux. В этой главе мы опишем только средства Sun RPC. Все три пакета RPC — Courier, Sun и DCE — похожи друг на друга, поскольку основные концепции RPC остались неизменны.

Большинство поставщиков Unix предоставляют отдельную подробную документацию средств RPC. Например, документация Sun доступна по адресу <http://docs.sun.com>, и в первом томе Developer Collection можно найти 280-страничное руководство разработчика «ONC+ Developer's Guide». Документация Digital Unix по адресу http://www.unix.digital.com/faqs/publications/pub_page/V40D_DOCS.HTM включает 116-страничный документ под заголовком «Programming with ONC RPC».

Сам по себе RPC вызывает противоречивые мнения. Восемь статей на эту тему можно найти по адресу <http://www.kohala.com/~rstevens/papers.others/rpc.comments.txt>.

В этой главе мы предполагаем наличие TI-RPC (независимая от протокола версия RPC) для большинства примеров и говорим только о протоколах TCP и UDP, хотя TI-RPC поддерживает все протоколы, какие только могут быть на данном узле.

16.2. Многопоточность

Вспомните листинг 15.6, где мы продемонстрировали автоматическое управление потоками, осуществляющее библиотекой дверей. При этом сервер по умолчанию являлся параллельным. Покажем теперь, что средства Sun RPC по умолчанию делают сервер последовательным. Начнем с примера из предыдущего раздела и изменим только процедуру сервера. В листинге 16.4 приведен текст новой функции, выводящей идентификатор потока, делающей 5-секундную паузу, выводящей идентификатор еще раз и завершающей работу.

Запустим сервер, а после этого запустим три экземпляра программы-клиента:

Хотя этого нельзя сказать по выводимому тексту, перед появлением очередного результата проходит примерно 5 секунд. Если мы посмотрим на текст, выводимый сервером, то увидим, что клиенты обрабатываются последовательно: сначала полностью обрабатывается запрос первого клиента, затем второго и третьего:

Один и тот же поток обслуживает все запросы клиентов. Сервер не является многопоточным по умолчанию.

ПРИМЕЧАНИЕ

Серверы дверей в главе 15 работали не в фоновом режиме, а запускались из интерпретатора. Это давало нам возможность добавлять отладочные вызовы `printf` в процедуры сервера. Однако серверы Sun RPC по умолчанию являются демонами и выполняют действия так, как это описано в разделе 12.4 [24]. Это требует вызова `syslog` из процедуры сервера для вывода диагностической информации. Однако мы указали флаг `-DDEBUG` при компиляции нашего сервера, что эквивалентно определению

в заглушке сервера (файле `square_svc.c`, создаваемом `grcsd`). Это запрещает функции `main` становиться демоном и оставляет ее подключенной к терминалу, в котором она была запущена. Поэтому мы можем спокойно вызывать `printf` из процедуры сервера.

Возможность создания многопоточного сервера появилась в Solaris 2.4 и реализуется добавлением параметра `-M` в строку вызова `grcsd`. Это делает код, создаваемый `grcsd`, защищенным. Другой параметр, `-A`, позволяет автоматически создавать потоки по мере необходимости для обслуживания запросов клиентов. Мы включаем оба параметра при вызове `grcsd`.

Однако для реализации многопоточности требуется внести изменения в текст клиента и сервера, чего мы могли ожидать, поскольку использовали тип `static` в листинге 16.3. Единственное изменение, которое нужно внести в файл `square.x`, — сменить номер версии с 1 на 2. В объявлениях аргументов процедуры и результатов ничего не изменится.

В листинге 16.5 приведен текст новой программы-клиента.

8 Мы объявляем переменную типа `square_out`, а не указатель на нее.

12-14 Вторым аргументом вызова `squaregros_2` становится указатель на переменную `out`, а последним аргументом является дескриптор клиента. Вместо возвращения указателя на результат (как в листинге 16.2) эта функция будет возвращать либо `RPC_SUCCESS`, либо некоторое другое значение в случае возникновения ошибок. Перечисление `enumclnt_stat` в заголовочном файле `<rpc/clnt_stat.h>` содержит все возможные коды ошибок.

В листинге 16.6 приведен текст новой процедуры сервера. Как и программа из листинга 16.4, эта версия выводит идентификатор потока, ждет 5 секунд, а затем завершает работу.

3-12 Требуемые для реализации многопоточности изменения включают изменение аргументов функций и возвращаемого значения. Вместо возвращения указателя на структуру результатов (как в листинге 16.3) указатель на эту структуру принимается в качестве второго аргумента функции. Указатель на структуру `svc_req` смешается на третью позицию. Теперь при успешном завершении функции возвращается значение `TRUE`, а при возникновении ошибок — `FALSE`.

13-19 Еще одно изменение заключается в добавлении функции, освобождающей все автоматически выделенные переменные. Эта функция вызывается из заглушки сервера после завершения работы

процедуры сервера и отправки результата клиенту. В нашем примере просто делается вызов подпрограммы `xdr_free` (о ней будет говориться более подробно в связи с листингом 16.19 и упражнением 16.10).

Если процедура сервера выделяла память под сохраняемый результат (например, в виде связного списка), этот вызов освободит занятую память.

Создадим программу-клиент и программу-сервер и запустим три экземпляра клиента одновременно:

На этот раз мы видим, что результаты выводятся одновременно, один за другим. Взглянув на выводимый сервером текст, отметим, что используются три серверных потока и все они выполняются одновременно:

ПРИМЕЧАНИЕ

Одним из печальных следствий изменений, требуемых для реализации многопоточности, является уменьшение количества систем, поддерживающих новый код. Например, в Digital Unix 4.0B и BSD/OS 3.1 используется старая система RPC, не поддерживающая многопоточность. Это означает, что если мы хотим компилировать и использовать нашу программу в системах обоих типов, нам нужно использовать условия `#ifdef` для обработки различий в вызовах клиента и сервера. Конечно, клиент в BSD/OS, не являющийся многопоточным, может вызвать процедуру многопоточного сервера в Solaris, но если мы хотим, чтобы клиент или сервер компилировался в обоих типах систем, исходный код нужно изменить, предусмотрев различия.

В описании листинга 16.5 мы достаточно бегло прошлись по действиям, выполняемым на нулевом этапе: регистрация сервера в локальной программе отображения портов и определение клиентом адреса порта не были разобраны детально. Отметим прежде всего, что на любом узле с сервером RPC должна выполняться программа port mapper (отображение портов). Этой программе присваивается адрес порта TCP 111 и UDP 111, и это единственное фиксированное значение портов Интернета для Sun RPC. Серверы RPC всегда связываются с временным портом, а затем регистрируют его в локальной службе отображения портов. После запуска клиент должен связаться с программой отображения портов, узнать номер временного порта сервера, а затем связаться с самим сервером через этот порт. Программа отображения портов предоставляет также службу имен, область действия которой ограничена системой.

ПРИМЕЧАНИЕ

Некоторые читатели могут возразить, что сетевая файловая система также имеет фиксированный номер порта 2049. Хотя во многих реализациях по умолчанию действительно используется именно этот порт, а в некоторых старых реализациях он вообще жестко «зашит» в клиентскую и серверную части NFS, большинство существующих реализаций позволяют использовать и другие порты. Большая часть клиентов NFS также связывается со службой отображения портов для получения номера порта.

В Solaris 2.x Sun переименовала службу отображения портов в RPCBIND. Причина этого изменения заключается в том, что термин «порт» подразумевает порт Интернета, тогда как пакет TI-RPC может работать с любым сетевым протоколом, а не только с TCP и UDP. Мы будем использовать традиционное название «программа отображения портов» (port mapper). Далее в этой главе мы будем подразумевать, что на данном узле поддерживаются только протоколы TCP и UDP.

Сервер и клиент работают следующим образом:

1. При переходе системы в многопользовательский режим запускается программа отображения портов. Исполняемый файл этого демона обычно называется portmap или gRPCbind.
2. При запуске сервера его функция main, являющаяся частью заглушки сервера, создаваемой gRPCbind, вызывает библиотечную функцию svc_create. Эта функция выясняет, какие сетевые протоколы поддерживаются узлом, и создает конечную точку (например, сокет) для каждого протокола, связывая временные порты с конечными точками протоколов TCP и UDP. Затем она связывается с локальной программой отображения портов для регистрации временных номеров портов TCP и UDP вместе с номером программы и номером версии.

Сама программа отображения портов также представляет собой программу RPC, и сервер регистрируется с помощью вызовов RPC (обращенных к известному порту 111). Описание процедур, поддерживаемых программой отображения портов, дается в стандарте RFC 1833 [20]. Существуют три версии этой программы RPC: вторая версия работает только с портами TCP и UDP, а версии 3 и 4 представляют собой новые версии, работающие по протоколу RPCBIND.

Можно получить список всех программ RPC, зарегистрированных в программе отображения портов, запустив программу gRPCinfo. Мы можем запустить эту программу, чтобы убедиться, что порт с номером 111 используется самой программой отображения портов:

(Мы исключили множество несущественных в данный момент строк вывода.) Мы видим, что Solaris 2.6 поддерживает все три версии протокола, все на порте 111, причем как TCP, так и UDP. Соответствие номеров программ RPC их именам обычно устанавливается в файле /etc/gRPC. Запустив ту же программу в BSD/OS 3.1, увидим, что в этой системе поддерживается только вторая версия программы отображения портов:

В Digital Unix 4.0B также поддерживается только вторая версия:

Затем процесс сервера приостанавливает работу, ожидая поступления запросов от клиентов. Это может быть новое соединение TCP или приход дей-таграммы UDP в порт UDP. Если мы запустим gRPCinfo после запуска сервера из листинга 16.3, мы увидим следующий результат:

где 824377344 соответствует 0x31230000 (номер программы, присвоенный ей в листинге 16.1). В том же листинге мы присвоили программе номер версии 1. Обратите внимание, что сервер готов принимать запросы от клиентов по протоколам TCP и UDP и клиент может выбирать, какой из этих протоколов он будет использовать при создании дескриптора клиента (последний аргумент clnt_create в листинге 16.2).

3. Клиент запускается и вызывает clnt_create. Аргументами (листинг 16.2) являются имя узла или IP-адрес сервера, номер программы, номер версии и строка, указывающая протокол связи. Запрос RPC направляется программе отображения портов узла сервера (для этого сообщения обычно используется протокол UDP), причем запрашивается информация об указанной версии указанной

программы с указанным протоколом. В случае успеха номер порта сохраняется в дескрипторе клиента для обработки всех последующих вызовов RPC через этот дескриптор.

В листинге 16.1 мы присвоили нашей программе номер 0x31230000. 32-разрядные номера программ подразделяются на группы, приведенные в табл. 16.1.

Таблица 16.1. Диапазоны номеров программ для Sun RPC

Программа grcinfo выводит список программ, зарегистрированных в системе. Другим источником информации о программах RPC могут являться файлы с расширением .x в каталоге /usr/include/rpcsvc.

Inetd и серверы RPC

По умолчанию серверы, созданные с помощью grcsen, могут вызываться сервером верхнего уровня inetd. Этот сервер описывается в разделе 12.5 [24]. Изучение содержимого заглушки сервера, создаваемой grcsen, показывает, что при запуске функции main сервера она проверяет, является ли стандартный поток ввода конечной точкой XTI, и если так, то предполагается, что сервер был запущен демоном inetd.

После создания сервера RPC, который будет вызываться inetd, следует добавить информацию об этом сервере в файл /etc/inetd.conf. Туда помещаются следующие данные: имя программы RPC, поддерживаемые номера программ, протоколы и полное имя исполняемого файла сервера. В качестве примера мы приводим строку из конфигурационного файла Solaris:

Первое поле содержит имя программы (которому будет сопоставлен номер с помощью файла /etc/grcs); поддерживаются версии 2, 3 и 4. Следующее поле задает конечную точку XTI (или сокет), третье поле говорит о том, что поддерживаются все протоколы видимых дейтаграмм. Если свериться с содержимым файла /etc/netconfig, мы узнаем, что таких протоколов два: UDP и /dev/clts. Глава 29 [24] описывает этот файл и адреса XTI. Четвертое поле (wait) указывает демону inetd на необходимость ожидания завершения этого сервера перед включением режима ожидания запроса клиента для конечной точки XTI. Все серверы RPC указывают атрибут wait в конфигурационном файле /etc/inetd.conf.

Следующее поле, root, указывает идентификатор пользователя, с которым будет выполняться программа. Последние два поля задают полное имя исполняемого файла программы и имя программы вместе с необходимыми аргументами командной строки (у данной программы они отсутствуют).

Демон inetd создаст конечные точки XTI и зарегистрирует их в программе отображения портов для соответствующих номеров программ и версий. Мы можем убедиться в этом с помощью grcinfo:

Четвертое поле содержит адреса XTI, причем $128 \times 256 + 11 = 32779$, и данное значение является временным номером порта, присвоенным этой концевой точке UDP.

Когда дейтаграмма UDP поступает в порт 32779, демон inetd обнаруживает готовность этой дейтаграммы к обработке и вызывает fork, а затем exec для запуска программы /usr/lib/netsvc/rstat/rpc.rstatd. Перед вызовами fork и exec концевая точка XTI будет скопирована в дескрипторы 0, 1 и 2, а все прочие дескрипторы inetd будут закрыты (рис. 12.7 [24]). Демон inetd также прекратит слушать эту конечную точку XTI, не реагируя на запросы пользователей до тех пор, пока сервер (дочерний процесс по отношению к inetd) не завершит работу. Это поведение определяется атрибутом wait.

Предположим, что эта программа была создана с помощью grcsen. Тогда она сможет распознать конечную точку XTI, подключенную к стандартному потоку ввода, и инициализировать ее как конечную точку сервера RPC. Это осуществляется вызовом функций RPC svc_tli_create и svc_reg, которые в данной книге не рассматриваются. Вторая функция (вопреки названию) не регистрирует сервер в программе отображения портов — это делается лишь однажды, при запуске сервера. Функция svc_gup прочитает пришедшую дейтаграмму и вызовет соответствующую процедуру сервера для обработки запроса клиента.

В обычной ситуации серверы, запускаемые демоном inetd, обрабатывают один запрос клиента и завершают работу, после чего inetd переходит в режим ожидания следующего запроса. Для оптимизации работы системы серверы RPC, созданные grcsen, ждут поступления нового запроса от клиента в течение некоторого времени (по умолчанию 2 минуты). В этом случае дейтаграмма обрабатывается уже запущенным сервером. Это исключает накладные расходы на вызов fork и exec при поступлении нескольких клиентских запросов подряд. По истечении периода ожидания сервер завершает работу, а демону inetd отсылается сигнал SIGCHLD, после чего он переходит в режим ожидания дейтаграмм по XTI.

16.4. Аутентификация

По умолчанию в запросе RPC не содержится информации о клиенте. Сервер отвечает на запрос клиента, не беспокоясь о том, что это за клиент. Это называется нулевой аутентификацией, или AUTH_NONE.

Следующий уровень проверки подлинности называется аутентификацией Unix, или AUTH_SYS. Клиент должен сообщить библиотеке RPC времени выполнения информацию о себе (имя узла, действующий идентификатор пользователя, действующий идентификатор группы, дополнительные идентификаторы группы) для включения в каждый запрос. Изменим программу из листинга 16.2 таким образом, чтобы она включала возможность осуществления аутентификации Unix. В листинге 16.7 приведен новый текст программы-клиента.

12-13 Эти строки были добавлены в данной версии программы. Сначала мы вызываем auth_destroy для удаления предыдущей аутентификационной информации, связанной с данным дескриптором клиента (то есть дескриптор нулевой аутентификации, создаваемый по умолчанию). Затем вызов authsys_create_default создает соответствующую аутентификационную структуру Unix и мы сохраняем ее в поле cl_auth структуры CLIENT. Оставшаяся часть клиента не претерпела изменений по сравнению с листингом 16.5.

В листинге 16.8 приведен текст процедуры сервера, измененный по сравнению с листингом 16.6. Мы не приводим текст процедуры square_prog_2_freeresult, которая не меняется.

6-8 Теперь мы используем указатель на структуру svc_req, которая всегда передается в качестве одного из аргументов процедуры сервера:

Поле rq_cred содержит неформатированную информацию о клиенте, а его поле oa_flavor содержит целое число, определяющее тип аутентификации. Термин «неформатированная» означает, что библиотека не обработала информацию, на которую указывает oa_base. Но если тип идентификации относится к одному из поддерживаемых библиотекой, то в готовой информации о клиенте, на которую указывает rq_clntcred, содержится некоторая структура, соответствующая данному типу аутентификации. Программа выводит тип аутентификации и проверяет, соответствует ли он AUTH_SYS.

Для аутентификации Unix указатель на готовую информацию (rq_clntcred) указывает на структуру authsys_parms, содержащую информацию о клиенте:

Мы получаем указатель на эту структуру и выводим имя узла клиента, его EUID и EGID.

Запустив сервер и один экземпляр клиента, посмотрим на выводимый сервером текст:

Аутентификация Unix используется редко, поскольку ее легко обойти. Мы можем легко построить собственные пакеты RPC, содержащие аутентификационную информацию в формате Unix, присвоив идентификатору пользователя и группы произвольные значения, и отправить их на сервер. Сервер никак не может проверить, те ли мы, кем представляемся.

ПРИМЕЧАНИЕ

Вообще-то NFS по умолчанию использует именно аутентификацию Unix, но запросы обычно отсылаются ядром клиента NFS через зарезервированный порт (раздел 2.7 [24]). Некоторые серверы NFS настроены так, чтобы отвечать только на запросы, поступающие по зарезервированному порту. Если вы доверяете узлу клиента подключение к своим файловым системам, вы доверяете и его ядру в том, что оно правильно предоставляет информацию о своих пользователях. Если сервер не требует подключения по резервному порту, хакеры могут написать свою собственную программу, которая будет посылать запросы серверу NFS с произвольным идентификатором пользователя. Даже если сервер требует подключения по зарезервированному порту, а у вас есть своя система, в которой вы обладаете правами привилегированного пользователя, и вы можете подключиться к сети, вы сможете отправлять свои собственные запросы с произвольным содержимым на сервер NFS.

Пакеты RPC — как запросы, так и ответы — содержат два поля, относящиеся к аутентификации: данные о пользователе и проверочную информацию (credentials, verifier). Примером такой структуры является документ с фотографией (паспорт, права и т. п.). Данные о пользователе соответствуют написанному в паспорте тексту (имя, адрес, дата рождения и т. п.), а проверочная

информация — это фотография. Существуют разные формы проверочной информации: фотография в данном случае полезнее, чем, например, рост, вес и пол. Если документ не содержит проверочной информации (как, например, читательский билет в библиотеке), любой может воспользоваться им и сказать, что он его владелец.

В случае нулевой аутентификации пакеты не содержат ни данных о пользователе, ни проверочной информации. В режиме аутентификации Unix данные о пользователе содержат имя узла, идентификаторы пользователя и группы, но поле проверочной информации пусто. Поддерживаются, однако, и другие формы аутентификации, для которых эти два поля содержат другую информацию.

- AUTH_SHORT — альтернативная форма аутентификации Unix, отправляемая сервером в поле verifier в ответ на запрос клиента. Она содержит меньшее количество информации, чем в режиме аутентификации Unix, и клиент может отсылать ее серверу при последующих запросах. Используется для уменьшения количества передаваемой по сети информации.
- AUTH_DES — аббревиатура DES означает Data Encryption Standard (стандарт шифрования данных). Эта форма аутентификации основана на использовании криптографии с секретным и открытым ключом. Эта схема также называется защищенным RPC (secure RPC), а если она используется в качестве основы для построения NFS, то такая NFS также называется защищенной.
- AUTH_KERB — эта схема основана на стандарте Kerberos института MIT.

В главе 19 книги [5] подробно рассказывается о двух последних формах аутентификации, включая их настройку и использование.

Рассмотрим стратегию обработки тайм-аутов и повторной передачи, используемую в средствах Sun RPC. Существуют два значения тайм-аутов:

1. Общий тайм-аут определяет время ожидания ответа сервера клиентом. Это значение используется протоколами TCP и UDP.
2. Тайм-аут повтора используется только UDP и определяет время ожидания между повторами запросов клиента, если ответ от сервера не приходит.

Для протокола TCP необходимость во введении тайм-аута повтора отсутствует, поскольку этот протокол является надежным. Если сервер не получает запроса от клиента, время ожидания по протоколу TCP со стороны клиента закончится и клиент повторит передачу. Когда сервер получает запрос клиента, он уведомляет об этом последний. Если уведомление о получении будет утрачено по пути к клиенту, тот должен будет еще раз переслать запрос. Повторные запросы сбрасываются сервером, но уведомления об их получении отсылаются клиенту. В надежных протоколах правильность доставки (время ожидания, повторная передача, обработка лишних копий данных и лишних уведомлений) обеспечивается на транспортном уровне и не входит в задачи библиотеки RPC. Один запрос, отправленный клиентом на уровне RPC, будет получен сервером ровно в одном экземпляре на уровне RPC. В противном случае клиент RPC получит сообщение о невозможности связаться с сервером. При этом совершенно не важно, что происходит на сетевом и транспортном уровнях.

После создания дескриптора клиента можно использовать функцию `clnt_control` для получения информации и изменения свойств клиента. Эта функция работает аналогично `fcntl` для дескрипторов файлов или `getsockopt` и `setsockopt` для сокетов:

Здесь `cl` представляет собой дескриптор клиента, а на что указывает `ptr` — зависит от значения `request`.

Изменим программу-клиент из листинга 16.2, добавив в нее вызов данной функции, и выведем значения тайм-аутов. В листинге 16.9 приведен текст новой программы-клиента.

10-12 Теперь протокол, являющийся последним аргументом `clnt_create`, указывается в качестве нового параметра командной строки.

13-14 Первым аргументом `clnt_control` является дескриптор клиента, вторым — тип запроса, а третьим — указатель на буфер. Наш первый запрос имеет значение `CLGET_TIMEOUT`; при этом возвращается значение общего тайм-аута в структуре `timeval`, адрес которой передается третьим аргументом. Этот запрос корректен для всех протоколов.

15-16 Следующий запрос имеет значение `CLGET_RETRY_TIMEOUT`. При этом должно возвращаться значение тайм-аута повтора, но этот запрос корректен только для протокола UDP. Следовательно, если функция возвращает значение `FALSE`, мы ничего не печатаем.

Изменим также и программу-сервер, добавив в нее ожидание продолжительностью 1000 секунд вместо 5, чтобы гарантировать получение тайм-аута по запросу клиента. Запустим сервер на узле `bsdi`, а клиент запустим дважды, один раз указав в качестве протокола TCP, а другой — UDP.

Результат будет не таким, как мы ожидали:

В случае с протоколом TCP значение тайм-аута, возвращенное `clnt_control`, было 30 секунд, но библиотека возвратила ошибку через 25 секунд. Для протокола UDP было получено значение общего тайм-аута -1.

Чтобы понять, что тут происходит, изучим текст заглушки клиента — функции `squareproc_1` в файле `square_clnt.c`, созданном `grcgen`. Эта функция вызывает библиотечную функцию с именем `clnt_call`, причем последним аргументом является структура типа `timeval` с именем `TIMEOUT`, объявляемая в этом файле, и инициализируется она значением 25 секунд. Этот аргумент `clnt_call` отменяет значение общего тайм-аута в 30 секунд для TCP и -1 для UDP. Он используется всегда, если клиент не устанавливает общий тайм-аут явно вызовом `clnt_control` с запросом `CLSET_TIMEOUT`. Если мы хотим изменить значение общего тайм-аута, следует вызывать `clnt_control`, а не изменять содержимое заглушки клиента.

ПРИМЕЧАНИЕ

Единственный способ проверить значение тайм-аута повтора для протокола UDP заключается в просмотре пакетов с помощью `tcpdump`. При этом можно увидеть, что первая дейтаграмма отправляется сразу после запуска клиента, а следующая — примерно 15 секунд спустя.

Управление соединением по TCP

Если мы будем наблюдать с помощью tcpdump за работой клиента и сервера из предыдущего примера, связывающихся по протоколу TCP, мы увидим, что сначала происходит установка соединения (трехэтапное рукопожатие TCP), затем отправляется запрос клиента и сервер отсылает уведомление о приеме этого запроса. Через 25 секунд после этого клиент отсылает серверу FIN, что вызвано завершением работы клиента, после чего следуют оставшиеся три этапа завершения соединения по TCP. В разделе 2.5 [24] эти этапы описаны подробно.

Мы хотим показать, что Sun RPC использует соединение по TCP следующим образом: новое соединение по TCP устанавливается при вызове `clnt_create` и оно используется для всех вызовов процедур, связанных с указанной программой и версией. Соединение по TCP завершается явно вызовом `clnt_destroy` или неявно по завершении процесса клиента:

Начнем с клиента из листинга 16.2 и изменим его, добавив второй вызов процедуры сервера, вызовы `clnt_destroy` и `pause`. В листинге 16.10 приведен текст новой программы-клиента.

После запуска получим ожидаемый результат:

Однако проверить наши предыдущие утверждения можно лишь с помощью результатов работы программы `tcpdump`. Она показывает, что создается одно соединение по TCP (вызовом `clnt_create`) и оно используется для обоих запросов клиента. Соединение завершается вызовом `clnt_destroy`, хотя клиент при этом и не завершает свою работу.

Идентификатор транзакций

Другая часть стратегии тайм-аутов и повторных передач заключается в использовании идентификаторов транзакций (transaction ID или XID) для распознавания запросов клиента и ответов сервера. Когда клиент вызывает функцию RPC, библиотека присваивает этому вызову 32-разрядный целочисленный номер и это значение отсылается в запросе RPC. Сервер должен добавить к своему ответу этот номер. При повторной отсылке запроса идентификатор не меняется. Служит он двум целям:

1. Клиент проверяет, что XID ответа совпадает с XID запроса. Если совпадения нет, ответ игнорируется. Если используется протокол TCP, у клиента практически нет шансов получить ответ с неправильным идентификатором, но при использовании протокола UDP поверх плохой сети вероятность получения неправильного XID достаточно высока.
2. Серверу разрешается помещать отсылаемые ответы в кэш, и для проверки идентичности ответов используется, в частности, именно XID. Об этом мы вскоре расскажем.

Пакет TI-RPC использует определенный алгоритм вычисления XID для нового запроса. Алгоритм этот описан ниже. Значок \wedge означает побитовую операцию XOR (исключающее ИЛИ):

Кэш повторных ответов

Для включения поддержки кэша повторных ответов в библиотеке RPC сервер должен вызывать функцию `svc_dg_enablecache`. После включения кэша выключить его нельзя, можно только запустить процесс заново:

Здесь `xprt` представляет собой транспортный дескриптор, являющийся полем структуры `svc_gref` (раздел 16.4). Адрес этой структуры является аргументом процедуры сервера. Размер определяет количество записей в выделяемом кэше.

Итак, эта функция включает поддержку кэширования всех отсылаемых ответов в очереди размером `size` записей. Каждый ответ однозначно определяется следующими параметрами:

- номером программы;
- номером версии;
- номером процедуры;
- XID;
- адресом клиента (IP-адрес + порт UDP).

При получении запроса клиента библиотека RPC ищет в кэше ответ на такой запрос. В случае его наличия ответ отсылается клиенту без повторного вызова процедуры сервера.

Цель использования кэша повторных ответов состоит в том, чтобы не нужно было вызывать процедуру сервера несколько раз при получении нескольких копий запроса клиента. Это может быть нужно в случае, если процедура неидемпотентна. Повторный запрос может быть получен из-за того, что ответ был утерян или у клиента время ожидания меньше, чем время передачи ответа по сети. Обратите внимание, что этот кэш действует только для протоколов, работающих с дейтаграммами (таких, как UDP), поскольку при использовании TCP повторный запрос никогда не может быть получен приложением — он будет обработан TCP (см. упражнение 16.6).

16.6. Семантика вызовов

В листинге 15.24 мы привели пример клиента интерфейса дверей, повторно отсылающего запрос на сервер при прерывании вызова `door_call` перехватываемым сигналом. Затем мы показали, что при этом процедура сервера вызывается дважды, а не однократно. Потом мы разделили процедуры сервера на две группы: идемпотентные, которые могут быть вызваны произвольное количество раз без возникновения ошибок, и неидемпотентные, наподобие вычитания определенной суммы из банковского счета.

Вызовы процедур могут быть разбиты на три группы:

1. «Ровно один раз» означает, что процедура была выполнена только один раз. Такого труждно достичь ввиду ненулевой вероятности сбоев в работе сервера.
2. «Не более одного раза» означает, что процедура вовсе не была выполнена или что она была выполнена один раз. Если вызвавшему процессу возвращается результат, мы знаем, что процедура была выполнена. Если процессу возвращается сообщение об ошибке, мы не знаем, была ли процедура выполнена хотя бы один раз или не была выполнена вовсе.
3. «По крайней мере один раз» означает, что процедура была выполнена один раз, а возможно, и больше. Это не вызывает проблем для идемпотентных процедур — клиент продолжает передавать запросы до тех пор, пока не получит правильный ответ. Однако если клиент отправит несколько запросов, существует вероятность, что процедура будет выполнена больше одного раза.

При возвращении из локальной процедуры мы можем быть уверены, что она была выполнена ровно один раз. Однако если процесс завершает работу после вызова процедуры, мы не знаем, успела она выполниться или нет. Для удаленных вызовов процедур возможно несколько ситуаций.

- Если используется протокол TCP и получен ответ, мы можем быть уверены, что удаленная процедура была вызвана ровно один раз. Однако если ответ не был получен (сервер вышел из строя), мы уже не можем сказать, была процедура выполнена или нет. Обеспечение семантики «ровно один раз» при учете возможности досрочного завершения работы сервера и неполадок в сети требует системы обработки транзакций, что лежит за границами возможностей RPC.
- Если используется UDP без серверного кэша и был получен ответ, мы можем быть уверены, что процедура была вызвана по крайней мере один раз, но возможно, и несколько.
- Если используется UDP с серверным кэшем и был получен ответ, мы можем быть уверены, что процедура была вызвана ровно один раз. Однако если ответ не был получен, мы оказывается в ситуации «не более одного раза» аналогично сценарию с TCP.

■ Если вы стоите перед выбором:

- TCP,
- UDP с кэшем повторных ответов,
- UDP без кэша повторных ответов —

мы можем порекомендовать следующее:

- всегда используйте TCP, если только для приложения не важны накладные расходы на обеспечение надежности;
- используйте систему обработки транзакций для неидемпотентных процедур, корректное выполнение которых важно (работа с банковскими счетами, бронирование авиабилетов и т. п.);
- для неидемпотентных процедур использование TCP предпочтительно по сравнению с UDP и кэшем, поскольку TCP был изначально ориентирован на надежность, а добавление кэша к приложению, использующему UDP, вряд ли даст то же самое, что и использование TCP (см., например, раздел 20.5 [24]);
- для идемпотентных процедур можно использовать UDP без кэша;
- для неидемпотентных процедур использование UDP без кэша опасно.

В следующем разделе будут рассмотрены дополнительные преимущества использования TCP.

Рассмотрим, что произойдет в случае досрочного завершения клиента или сервера при использовании транспортного протокола TCP. Поскольку протокол UDP не подразумевает установку соединения, при завершении процесса его собеседнику не отсылается никаких сообщений. При завершении работы одного из процессов второй дождется тайм-аута, после чего, возможно, повторно отошлет запрос и наконец прекратит попытки, выдав сообщение об ошибке, как показывалось в предыдущем разделе. При завершении работы процесса, установившего соединение по TCP, это соединение завершается отправкой пакета FIN [24, с. 36-37], и мы хотим узнать, что делает библиотека RPC при получении этого пакета.

Досрочное завершение сервера

Завершим работу сервера досрочно, в процессе обработки запроса клиента. Единственное изменение в программе-клиенте будет заключаться в удалении аргумента `tcp` из вызова `clnt_call` в листинге 16.2 и включении протокола в набор аргументов командной строки, как в листинге 16.9. В процедуру сервера мы добавим вызов `abort`. Это приведет к завершению работы процесса-сервера и отправке пакета FIN клиенту, что мы можем проверить с помощью `tcpdump`.

Запустим в системе Solaris клиент для сервера, работающего под BSD/OS:

В момент получения клиентом пакета FIN библиотека RPC находилась в состоянии ожидания ответа сервера. Она получила неожиданный ответ и вернула ошибку в вызове `squaregoc_1`. Ошибка (`RPC_CANTRECV`) сохраняется библиотекой в дескрипторе клиента, и вызов `clnt_sperror` (из функции-обертки `Clns_create`) при этом печатает сообщение `Unable to receive`. Оставшаяся часть сообщения об ошибке (`An event requires attention`) соответствует ошибке XTI, сохраненной библиотекой, которая также выводится `clnt_sperror`. Вызов удаленной процедуры может вернуть одну из примерно 30 различных ошибок `RPC_xxx`. Все они перечислены в заголовочном файле `<rpc/clnt_stat.h>`.

Если мы поменяем клиент и сервер местами, мы увидим то же сообщение об ошибке, возвращаемое библиотекой RPC (`RPC_CANTRECV`), но при этом будет выведено дополнительное сообщение:

Сервер в Solaris не был скомпилирован как многопоточный, и когда мы вызвали `abort`, была завершена работа всего процесса. Если мы запустим многопоточный сервер и завершим работу только одного потока — того, который обслуживает данный запрос клиента, — все изменится. Чтобы продемонстрировать это, заменим вызов `abort` на `pthread_exit`, как мы сделали в программе из листинга 15.20. Запустим клиент в BSD/OS, а многопоточный сервер — в Solaris:

После завершения работы потока сервера соединение с клиентом по TCP не разрывается. Оно остается открытым, поэтому клиенту не отсылается пакет FIN. Клиент выходит по тайм-ауту. Мы увидели бы то же сообщение об ошибке, если бы узел, на котором находится сервер, прекратил работу после получения запроса от клиента и отправки уведомления.

Досрочное завершение клиента

Если клиент, использующий TCP, завершает работу в процессе выполнения процедуры RPC, серверу отправляется пакет FIN. Мы хотим узнать, как библиотека сервера реагирует на этот пакет и уведомляет об этом процедуру сервера. (В разделе 15.11 мы говорили, что поток сервера дверей отменяется при досрочном завершении клиента.)

Чтобы сымитировать такую ситуацию, клиент вызывает `alarm(3)` непосредственно перед вызовом процедуры сервера, а процедура сервера вызывает `sleep(6)`. Так же мы поступили и в нашем примере с дверьми в листингах 15.25 и 15.26. Поскольку клиент не перехватывает сигнал `SIGALRM`, процесс завершается ядром примерно за 3 секунды до отправки ответа серверу. Запустим клиент в BSD/OS, а сервер в Solaris:

Случилось то, что мы и ожидали. А вот на сервере не происходит ничего необычного. Процедура сервера благополучно заканчивает 6-секундную паузу и возвращается. Если мы взглянем на передаваемую по сети информацию с помощью `tcpdump`, мы увидим следующее:

- при завершении работы клиента (через 3 секунды после запуска) серверу отправляется пакет FIN, и сервер высылает уведомление о его приеме. В TCP для этого используется термин half-close (наполовину закрытое соединение, раздел 18.5 [22]);
- через 6 секунд после запуска клиента сервер отсылает ответ, который переправляется клиенту протоколом TCP. По соединению TCP можно отправлять данные после получения FIN, поскольку соединения TCP являются двусторонними, о чем говорится в книге [24, с. 130-132]. Клиент отвечает пакетом RST, поскольку он уже завершил работу. Он будет получен сервером при следующем открытии этого соединения, но это ни к чему не приведет.

Подведем итоги.

- При использовании UDP клиенты и серверы RPC не имеют возможности узнать о досрочном завершении одного из них. Они могут выходить по тайм-ауту, если ответ не приходит, но тип ошибки при этом определить не удастся: причина может быть в досрочном завершении процесса, сбое узла, недоступности сети и т. д.
- Клиент RPC, использующий TCP, может узнать о возникших на сервере проблемах, поскольку при досрочном завершении сервера его конец соединения автоматически закрывается. Это, однако, не помогает, если сервер является многопоточным, поскольку такой сервер не закрывает соединение в случае отмены потока с процедурой сервера. Мы также не получаем информации в случае сбоя узла, поскольку при этом соединение TCP не закрывается. Во всех этих случаях следует использовать выход по тайм-ауту.

В предыдущей главе мы использовали двери для вызова процедуры одного процесса из другого процесса. При этом оба процесса выполнялись на одном узле, поэтому необходимости в преобразовании данных не возникало. Однако RPC используется для вызова процедур на разных узлах, которые могут иметь различный формат хранения данных. Прежде всего могут отличаться размеры фундаментальных типов (в некоторых системах long имеет длину 32 бита, а в других — 64). Кроме того, может отличаться порядок битов (big-endian и little-endian, о чем говорится в книге [24, с. 66-69 и 137-140]. Мы уже столкнулись с этой проблемой, когда обсуждали листинг 16.3. Сервер у нас работал на компьютере с little-endian x86, а клиент — на big-endian Sparc, но мы могли без проблем обмениваться данными (в нашем примере — одно длинное целое).

Sun RPC использует стандарт XDR (External Data Representation — представление внешних данных) для описания и кодирования данных (RFC 1832 [19]). XDR является одновременно языком описания данных и набором правил для их кодирования. В XDR используется скрытая типизация (*implicit typing*), то есть отправитель и получатель должны заранее знать тип и порядок данных. Например, два 32-разрядных целых, одно число с плавающей точкой и одинарной точностью и строка символов.

ПРИМЕЧАНИЕ

Приведем сравнение из мира OSI. Для описания данных обычно используется нотация ASN.1 (Abstract Syntax Notation one), а для кодирования — BER (Basic Encoding Rules). Эта схема также использует явную типизацию, то есть перед каждым значением указывается его тип. В нашем примере поток байтов содержал бы: спецификатор типа целого, целое, спецификатор типа целого, целое, спецификатор типа single, число с плавающей точкой и одинарной точностью, спецификатор типа строки символов, строку символов.

Представление всех типов согласно XDR требует количества байтов, кратного четырем. Эти байты всегда передаются в порядке big-endian. Целые числа со знаком передаются в дополнительном коде, а числа с плавающей точкой передаются в формате IEEE. Поля переменной длины могут содержать до 3 байтов дополнения в конце, так чтобы подогнать начало следующего элемента до адреса, кратного четырем. Например, 5-символьная строка ASCII будет передана как 12 байтов:

- 4-байтовое целое, содержащее значение 5;
- 5-байтовая строка;
- 3 байта со значением 0 (дополнение).

При описании XDR и поддерживаемых типов данных следует уточнить три момента.

1. Как объявляются переменные различных типов в файле спецификации RPC (файл с расширением .x)? В наших примерах пока что использовалось только длинное целое.
2. В какой тип языка С преобразуется данный тип программой grcsdgen при составлении заголовочного файла?
3. Каков реальный формат передаваемых данных?

Таблица 16.2 содержит ответы на первых два вопроса. Для составления этой таблицы мы создали файл спецификации RPC со всеми поддерживаемыми стандартом XDR типами. Этот файл был обработан grcsdgen, после чего мы изучили получившийся заголовочный файл.

Таблица 16.2. Типы данных, поддерживаемые `xdr` и `grcsdgen`

Опишем содержимое таблицы более подробно.

1. Декларация `const` преобразуется в `#define`.
2. Декларация `typedef` преобразуется в `typedef`.
3. Пять целых типов со знаком. Передаются XDR как 32-разрядные значения (первые четыре типа), а последний — как 64-разрядное.

ПРИМЕЧАНИЕ

64-разрядное целое поддерживается многими компиляторами С как формат `long long int` или просто `long long`. Многие, но не все компиляторы и операционные системы поддерживают такой формат. Поскольку в созданном заголовочном файле объявляются переменные типа `longlong_t`, в другом заголовочном файле должно содержаться следующее определение:

Длинное целое в XDR занимает 32 бита, но длинное целое языка С в 64-разрядных системах Unix может занимать и 64 бита (например, в модели LP64, описанной в [24, с. 27]). Имена формата XDR устарели лет на десять и не слишком соответствуют современным стандартам. Лучше, если бы они назывались как-нибудь вроде `int8_t`, `int16_t` и т. д.

4. Пять целых типов без знака. Первые 4 передаются как 32-разрядные значения, а последнее — как 64-разрядное.

5. Три типа данных с плавающей точкой. Первый передается как 32-разрядное значение, второй — как 64-разрядное, а третий — как 128-разрядное.

ПРИМЕЧАНИЕ

Четверная точность для чисел с плавающей точкой (quadruple precision) поддерживается в С для типов `long double`. Не все компиляторы и операционные системы его воспринимают. Ваш компилятор может пропустить `long double`, но работать с этой переменной как с `double`. Поскольку в созданном заголовочном файле объявляются переменные типа `quadruple`, нужно создать другой заголовочный файл с объявлением

В Solaris 2.6, например, нам пришлось бы включить строку

в начало файла спецификации RPC, потому что этот заголовочный файл содержит требуемое определение. Знак процента перед `#include` говорит программе `grcsen` о необходимости поместить остаток строки непосредственно в создаваемый заголовочный файл.

6. Тип `boolean` эквивалентен целому со знаком. Заголовки RPC также определяют константу `TRUE` равной 1, а `FALSE` равной 0.

7. Перечисление (enumeration) эквивалентно целому со знаком и совпадает с типом данных `enum` в С. `grcsen` также создает определение типа для данной переменной.

8. Скрытые данные фиксированной длины передаются библиотекой как 8-разрядные значения без интерпретации.

9. Скрытые данные переменной длины также представляют собой последовательность неинтерпретируемых данных, но количество реально передаваемых данных помещается в целочисленную переменную и посыпается перед самими данными. При отправке данных такого типа (например, при заполнении списка аргументов перед вызовом RPC) следует указать длину, прежде чем делать вызов. При приеме данного типа данных следует выяснить значение длины, чтобы определить, сколько данных будет принято.

10. Стока представляет собой последовательность ASCII-символов. В памяти строка хранится как обычная строка символов языка С, завершаемая нулем, но при передаче перед ней отправляется целое без знака, в которое помещается количество символов данной строки (без завершающего нуля). При отправке данных такого типа размер строки определяется библиотекой с помощью вызова `strlen`. При приеме данные такого типа помещаются в строку символов С, завершающую нулем.

11. Массив фиксированной длины любого типа передается как последовательность `n` элементов данного типа.

12. Массив переменной длины любого типа передается как целое без знака, указывающее количество элементов, и последовательность элементов данного типа. Максимальное количество элементов в объявлении может быть опущено. Но если это количество указать при компиляции программы, библиотека будет проверять, не превосходит ли реальная длина указанного значения `m`.

13. Структура передается как последовательность полей. `grcsen` также создает определение типа для данного имени переменной (`typedef`).

14. Размеченное объединение состоит из целочисленного дискриминанта и набора типов данных (ветвей), зависящих от значения дискриминанта. В табл. 16.2 мы показываем, что дискриминант должен быть типа `int`, но он может быть и `unsigned int`, и `enum`, и `bool` (все эти типы передаются как 32-разрядные целые). При передаче размеченного объединения передается 32-разрядное значение дискриминанта, за которым следует значение той ветви, которая ему соответствует. В ветви `default` часто объявляется тип `void`, что означает отсутствие передаваемой вслед за дискриминантом информации. Ниже мы продемонстрируем это на примере.

15. Дополнительные данные представляют собой специальный тип объединения, описанный в примере из листинга 16.24. Объявление XDR выглядит как объявление указателя в языке С, и именно указатель объявляется в созданном заголовочном файле.

На рис. 16.3 сведена информация о кодировании различных типов данных в XDR.



Рис. 16.3. Кодирование типов данных в XDR

Пример: использование XDR без RPC

Приведем пример использования XDR без RPC. Мы воспользуемся стандартом XDR для кодирования структуры данных в машинно-независимое представление, в котором они могут быть обработаны другими системами. Этот метод может использоваться для написания файлов или для отправки данных по сети в машинно-независимом формате. В листинге 16.11 приведен текст файла спецификации data.x, который на самом деле является файлом спецификации XDR, поскольку мы не объявляем никаких процедур RPC.

ПРИМЕЧАНИЕ

Суффикс имени файла (.x) происходит от термина «файл спецификации XDR». Спецификация RPC утверждает, что язык RPC (RPCL) идентичен XDR в части, относящейся к описанию данных. В RPCL была добавлена только возможность описания процедур.

1-11 Мы объявляем перечислимый тип с двумя значениями и размеченное объединение, использующее это перечисление в качестве дискриминанта. Если дискриминант имеет значение RESULT_INT, после значения дискриминанта передается целое число. Если дискриминант имеет значение RESULT_DOUBLE, за ним передается число с плавающей точкой двойной точности. В противном случае после дискриминанта не передается ничего.

12-21 Мы объявляем структуру, состоящую из различных типов, поддерживаемых XDR.

Поскольку мы не объявляем процедур RPC, программа grscen не создаст заглушку клиента и заглушку сервера. Однако она создаст заголовочный файл data.h и файл data_xdr.c, содержащий функции XDR, обеспечивающие кодирование и декодирование данных, объявленных в файле data.x.

В листинге 16.12 приведен получающийся в результате работы grscen заголовочный файл data.h. Содержимое этого файла выглядит так, как мы и предполагали (табл. 16.2).

В файле data_xdr.c объявляется функция `xdr_data`, вызываемая для кодирования и декодирования структуры `data`, которую мы определили. Суффикс имени функции `_data` соответствует имени нашей структуры из листинга 16.11. Первая программа, которую мы напишем, будет называться `write.c`. Она будет присваивать значения полям структуры `data`, вызывать `xdr_data` для кодирования всех полей в формат XDR и записывать результат в стандартный поток вывода.

Эта программа приведена в листинге 16.13.

12-32 Сначала мы присваиваем полям структуры ненулевые значения. В случае полей переменной длины мы должны установить длину этих полей. Мы присваиваем дискриминанту размеченного объединения значение `RESULT_INT` и помещаем в его соответствующее поле значение 123.

33 Мы вызываем `malloc` для выделения буфера, в который подпрограммы XDR будут помещать результаты своей работы. Адрес и размер буфера должны быть кратны четырем. Выделение массива `char` не гарантирует этого.

34 Функция библиотеки времени выполнения `xdrmem_create` инициализирует буфер, на который указывает `buff`, предназначенный для использования функциями XDR как поток в памяти. Мы выделяем переменную типа XDR с именем `xhandle` и передаем адрес этой переменной в качестве первого аргумента. Библиотека XDR времени выполнения хранит в этой переменной всю необходимую информацию (указатель на буфер, текущее положение в буфере и т. п.). Последний аргумент имеет значение `XDR_ENCODE`, что указывает XDR на необходимость преобразования данных из формата узла в формат XDR.

35-36 Мы вызываем функцию `xdr_data`, созданную `grcgen` в файле `data_xdr.c`, и она кодирует структуру `out` в формат XDR. Возвращаемое значение `TRUE` говорит об успешном завершении работы функции.

37-38 Функция `xdr_getpos` возвращает текущее положение библиотеки XDR в выходном буфере (то есть сдвиг байта, в который будут помещены очередные данные). Его мы трактуем как размер готовых к записи данных.

В листинге 16.14 приведен текст программы `read`, которая считывает данные из файла, записанного предыдущей программой, и выводит значения всех полей структуры `data`.

11-13 Вызывается функция `malloc` для выделения буфера. В этот буфер считывается файл, созданный предыдущей программой.

14-17 Инициализируем поток XDR, указав флаг `XDR_DECODE`, означающий, что преобразование производится из формата XDR в формат узла. Мы инициализируем структуру `i` нулями и вызываем `xdr_data` для декодирования содержимого буфера `buff` в эту структуру. Мы обязаны инициализировать принимающую структуру нулями, поскольку некоторые из подпрограмм XDR (например, `xdr_string`) требуют выполнения этого условия. `xdr_data` — это та же функция, которую мы вызывали в листинге 16.13. Изменился только последний аргумент `xdrmem_create`: в предыдущей программе мы указывали `XDR_ENCODE`, а в этой — `XDR_DECODE`. Это значение сохраняется в дескрипторе XDR (`xhandle`) функцией `xdrmem_create` и затем используется библиотекой XDR для выбора между кодированием и декодированием данных.

18-42 Мы выводим значения всех полей структуры `data`.

43 Для освобождения памяти мы вызываем функцию `xdr_free` (см. упражнение 16.10).

Запустим программу `write` на компьютере Sparc, перенаправив стандартный вывод в файл с именем `data`:

Мы видим, что размер файла равен 72 байтам что соответствует рис. 16.4, на котором изображена схема хранения данных.

Прочитав этот файл в BSD/OS или Digital Unix, мы получим те результаты, на которые и рассчитывали:



Рис. 16.4. Формат потока XDR, записанный в листинге 16.13

Пример: вычисление размера буфера

В предыдущем примере мы выделяли буфер размера BUFSIZE (определенного в файле uprirc.h в листинге B.1), и этого было достаточно. К сожалению, не существует простого способа вычислить объем памяти, нужный XDR для кодирования конкретных данных. Вычислить размер структуры вызовом sizeof недостаточно, потому что каждое поле кодируется XDR по отдельности. Нам придется перебирать элементы структуры, прибавляя к конечному результату объем памяти, нужный XDR для кодирования очередного элемента. В листинге 16.15 приведен пример простой структуры с тремя полями.

Программа, текст которой приведен в листинге 16.16, вычисляет размер буфера, требуемого XDR для кодирования этой структуры. Он получается равным 28 байт.

8-9 Макрос RNDUP определен в файле <grc/xdr.h>. Он округляет аргумент к ближайшему кратному BYTES_PER_XDR_UNIT (4). Для массива фиксированного размера вычисляется размер каждого элемента, который затем умножается на количество элементов.

Проблема возникает в случае использования типов данных переменной длины. Если мы объявим stringd<10>, максимальный размер будет RNDUP(sizeof(int)) (для длины) плюс RNDUP(sizeof(char)*10) (для символов строки). Но мы не можем вычислить размер буфера, если максимальный размер не указан в объявлении переменной (например, float e<>). Лучше всего в этом случае выделять буфер с запасом, а потом проверять, не возвращают ли подпрограммы XDR ошибку (упражнение 16.5).

Пример: необязательные данные

Существуют три способа задания необязательных данных в файле XDR, примеры для всех приведены в листинге 16.17.

1-8 Мы определяем объединение с ветвями FALSE и TRUE и структуру этого типа. Если флаг дискриминанта TRUE, за ним следует значение типа long; в противном случае за ним ничего не следует. После кодирования библиотекой XDR это объединение будет закодировано как:

- 4 байта флага со значением 1 (TRUE) и 4 байта целочисленного значения либо
- 4 байта флага со значением 0 (FALSE).

9 Если мы указываем массив переменной длины с одним возможным элементом, он будет передан как:

- 4 байта со значением 1 и 4 байта значения либо
- 4 байта со значением 0.

10 Новый способ определения необязательных данных заключается в объявлении указателя. Он будет закодирован как:

- 4 байта со значением 1 и 4 байта значения либо
- 4 байта со значением 0

в зависимости от значения соответствующего указателя при кодировании данных. Если указатель ненулевой, используется первый вариант кодирования. Если указатель нулевой, получится второй вариант. Это удобный способ кодирования необязательных данных в случае, если в нашем коде имеется указатель на эти данные.

Важная деталь реализации, благодаря которой оба варианта дают одинаковый результат при кодировании, заключается в том, что значение TRUE равно 1, что совпадает с длиной массива переменной длины, когда в нем есть один элемент.

В листинге 16.18 приведен текст заголовочного файла, созданного программой grcsen для данного файла спецификации.

14-21 Хотя все три аргумента кодируются одинаково, способы присваивания и получения их значений в языке С различны.

В листинге 16.19 приведен текст простой программы, устанавливающей значения всех трех аргументов так, что ни одно из полей long не кодируется.

12-14 Дискриминанту объединения присваивается значение FALSE, длина массива переменной длины устанавливается в 0, а указатель делается нулевым (NULL).

15-19 Мы выделяем буфер и кодируем структуру out в поток XDR.

20-22 Мы выводим содержимое буфера XDR по 4 байта, используя функцию ntohl (host-to-network long integer) для преобразования из порядка XDR big-endian в байтовый порядок узла. В результате получается именно то, что должно было быть помещено в буфер библиотекой XDR времени выполнения:

Как мы и предполагали, каждому аргументу отводится 4 байта со значением 0, указывающим на то, что за ним не следует никаких данных.

В листинге 16.20 приведена измененная версия программы, которая присваивает значения всем трем аргументам, кодирует их в поток XDR и выводит его содержимое.

12-18 Для присваивания значения объединению мы устанавливаем дискриминант в TRUE, а затем присваиваем значение полю long. Длину массива мы также сначала устанавливаем в 1. Указатель мы устанавливаем на соответствующее значение в памяти.

При запуске этой программы мы получим ожидаемые шесть 4-байтовых значений:

Пример: обработка связного списка

Если осуществляется передача необязательных данных, мы можем расширить возможности указателей в XDR и использовать их для кодирования и декодирования связных списков, содержащих произвольное количество элементов. В нашем примере используется связный список пар имя-значение. Соответствующий файл спецификации XDR приведен в листинге 16.21.

1-5 Структура mylist содержит одну пару имя-значение и указатель на следующую структуру такого типа. Указатель в последней структуре списка будет нулевым.

В листинге 16.22 приведен текст заголовочного файла, созданного программой grcsen из файла opt2.x.

В листинге 16.23 приведен текст программы, инициализирующей связный список с тремя парами имя-значение и кодирующей его с помощью библиотеки XDR.

11-22 Мы выделяем память под четыре элемента, но инициализируем только три из них. Первая запись nameval[2], потом nameval[1] и nameval[0]. Указатель на начало списка (out.list) устанавливается на &nameval[2]. Мы инициализируем список в таком порядке, чтобы показать, что библиотека XDR обрабатывает указатели и порядок в списке оказывается именно таким, каким он был в нашей программе, и не зависит от того, какие массивы для этого используются. Мы также инициализируем значения элементов списка шестнадцатеричными величинами, поскольку будем выводить их в этом формате.

Вывод программы показывает, что перед каждым элементом списка идет значение 1 в 4 байтах (что мы можем считать длиной массива переменной длины с одним элементом или булевским значением TRUE). Четвертая запись состоит из 4 байт, в которых записан 0. Она обозначает конец списка:

При декодировании списка библиотека XDR будет динамически выделять память под его элементы и указатели и связывать все это вместе, что позволит легко переходить от одного элемента списка к другому в программе на C.

16.9. Форматы пакетов RPC

На рис. 16.5 приведен формат запроса RPC в пакете TCP.

Поскольку TCP передает поток байтов и не предусматривает границ сообщений, приложение должно предусматривать способ разграничения сообщений. Sun RPC определяет запись как запрос или ответ, и каждая запись состоит из одного или более фрагментов. Каждый фрагмент начинается с 4-байтового значения: старший бит является флагом последнего фрагмента, а следующие 31 бит представляют собой счетчик (длина фрагмента). Если бит последнего фрагмента имеет значение 0, данный фрагмент не является последним в записи.

ПРИМЕЧАНИЕ

Это 4-байтовое значение передается в порядке big-endian, так же как и 4-байтовые целые в XDR, но оно не относится к стандарту XDR, поскольку он не предусматривает передачи битовых полей.

Если вместо TCP используется UDP, первое поле в заголовке UDP будет идентификатором транзакции (XID), как показано на рис. 16.7.

ПРИМЕЧАНИЕ

При использовании TCP на размер запроса и ответа RPC ограничения не накладываются, поскольку может использоваться любое количество фрагментов, а длина каждого из них задается 31-разрядным целым. Но при использовании протокола UDP запрос (и ответ) должен помещаться в дейтаграмму целиком, а максимальное количество данных в ней — 65507 байт (для IPv4). Во многих реализациях, предшествовавших TI-RPC, размер ограничивался значением около 8192 байт, поэтому если требуется передавать более 8000 байт, следует пользоваться протоколом TCP.



Рис. 16.5. Запрос RPC в пакете TCP

Приведем спецификацию XDR для запроса RPC, взятую из RFC 1831. Имена на рис. 16.5 взяты из этой спецификации:

Содержимое скрытых данных переменной длины, содержащих сведения о пользователе и проверочную информацию, зависит от типа аутентификации. Для нулевой аутентификации, используемой по умолчанию, длина этих данных должна быть установлена в 0. Для аутентификации Unix эти данные содержат следующую структуру:

Если тип аутентификации AUTH_SYS, тип проверки должен быть AUTH_NONE. Формат ответа RPC сложнее, чем формат запроса, поскольку в нем могут передаваться сообщения об ошибках. На рис. 16.6 показаны возможные варианты. На рис. 16.7 показан формат ответа RPC в случае успешного выполнения процедуры. Ответ передается по протоколу UDP.

Ниже приводится текст спецификации XDR ответа RPC, взятый из RFC 1831.



Рис. 16.6. Возможные варианты ответов RPC

Вызов может быть отклонен сервером, если номер версии RPC не тот или возникает ошибка аутентификации:



Рис. 16.7. Ответ на успешно обработанный вызов в дейтаграмме UDP

16.10. Резюме

Средства Sun RPC дают возможность создавать распределенные приложения, в которых клиентская часть может выполняться на одном узле, а серверная — на другом. Сначала следует определить процедуры сервера, которые могут быть вызваны клиентом, и написать файл спецификации RPC, описывающий аргументы и возвращаемые значения этих процедур. Затем пишется функция `main` клиента,зывающая процедуры сервера, а потом сами эти процедуры. Программа клиента выглядит так, как будто она просто вызывает процедуры сервера, но на самом деле их скрытое взаимодействие по сети обеспечивается библиотекой RPC.

Программа `grcsdep` является краеугольным камнем приложения, использующего RPC. Она считывает файл спецификации и создает заглушку клиента и заглушку сервера, а также функции, вызывающие требуемые подпрограммы XDR, которые осуществляют все необходимые преобразования данных. Библиотека XDR также является важной частью процесса. XDR определяет стандарт обмена данными различного формата между разными системами, у которых может быть по-разному определен, например, размер целого, порядок байтов и т. п. Как мы показали, XDR можно использовать и отдельно от RPC для обмена данными в машинно-независимом стандартном формате. Для передачи данных можно использовать любой механизм (сокеты, XTI, диски, компакт-диски или что угодно).

В Sun RPC используется свой стандарт именования программ. Каждой программе присваивается 32-разрядный номер программы, 32-разрядный номер версии и 32-разрядный номер процедуры. Каждый узел с сервером RPC должен выполнять программу отображения портов в фоновом режиме (`RPCBIND`). Серверы RPC привязывают временные порты TCP и UDP к своим процедурам, а затем регистрируют эти порты в программе отображения портов, указывая номера программ и версий. При запуске клиент RPC связывается с программой отображения портов узла, где запущен сервер RPC, и выясняет номер нужного ему порта, а затем связывается с самим сервером по протоколам TCP или UDP.

По умолчанию клиенты RPC не предоставляют аутентификационной информации и серверы RPC обрабатывают все приходящие запросы. Это аналогично написанию собственного приложения клиент-сервер с использованием сокетов или XTI. В Sun RPC предоставляются три дополнительные формы аутентификации: аутентификация Unix (предоставляется имя узла клиента, идентификатор пользователя и группы), аутентификация DES (основанная на криптографии с секретным и открытым ключом) и аутентификация Kerberos.

Понимание стратегии тайм-аутов и повторных передач пакета RPC важно при использовании RPC (как и любой формы сетевого программирования). При использовании надежного транспортного протокола, такого, как TCP, клиенту RPC нужно использовать только общий тайм-аут, поскольку потеря пакетов или прием лишних копий целиком обрабатываются на транспортном уровне. Когда используется ненадежный транспортный протокол, такой как UDP, пакет RPC использует тайм-аут повтора в дополнение к общему тайм-ауту. Идентификатор транзакций используется клиентом RPC для проверки ответа на соответствие отправленному запросу.

Любой вызов процедуры может быть отнесен к группе «ровно один», «не более чем один» или «не менее чем один». Для локальных вызовов этот вопрос можно не принимать во внимание, но при использовании RPC эту разницу следует понимать. Следует также понимать различия между идемпотентными и неидемпотентными процедурами.

Sun RPC — это большой пакет, и мы лишь вкратце обсудили особенности его использования. Тем не менее сведений, приведенных в этой главе, должно быть достаточно для написания приложений целиком. Использование `grcsdep` скрывает многие детали и упрощает кодирование. Документация Sun описывает различные уровни кодирования RPC — упрощенный интерфейс, верхний уровень, средний уровень, уровень экспертов и низкий уровень, но эти категории достаточно бессмысленны. Количество функций в библиотеке RPC — 164, они могут быть разделены следующим образом:

- 11 `auth_` (аутентификация);
- 26 `clnt_` (клиентские);
- 5 `rmap_` (доступ к программе отображения портов);
- 24 `rpc_` (общего назначения);
- 44 `svc_` (серверные);
- 54 `xdr_` (преобразования XDR).

Для сравнения отметим, что интерфейсы сокетов и XTI содержат по 25 функций, а интерфейсы дверей, очередей сообщений Posix и System V, семафоров и разделяемой памяти содержат по 10

функций. 15 функций используются для работы с потоками в стандарте Posix, 10 — для работы с условными переменными. 11 функций используются для работы с блокировками чтения-записи Posix и одна при работе с блокировкой записей fcntl.

Упражнения

1. При запуске сервер регистрируется в программе отображения портов. Что происходит при завершении сервера, например, клавишей завершения программы с терминала? Что произойдет, если на этот сервер впоследствии придет запрос от клиента?
2. Клиент взаимодействует с сервером RPC по протоколу UDP, и кэш не включен. Клиент посыпает запрос на сервер, но серверу требуется 20 секунд до отправки ответа. Клиент посыпает запрос повторно через 15 секунд, что приводит к повторному запуску процедуры сервера. Что произойдет со вторым ответом сервера?
3. Тип XDR `string` всегда кодируется в значение длины и последовательность символов. Что произойдет, если мы напишем `char c[10]` вместо `string s<10>`?
4. Измените максимальный размер `string` в листинге 16.11 со 128 на 10 и запустите программу `write`. Что произойдет? Уберите ограничение длины и сравните файл `data_xdr.c` с тем, который был создан, когда длина была ограничена. Что изменилось?
5. Измените третий аргумент в вызове `xdrmem_create` (размер буфера) в листинге 16.13 на 50 и посмотрите, что произойдет.
6. В разделе 16.5 мы описали включение кэша повторных ответов при использовании протокола UDP. Мы можем сказать, что протокол TCP имеет свой собственный кэш такого рода. О чём мы говорим и как велик этот кэш у протокола TCP? (Подсказка: как протокол TCP определяет, что принятая копия полученных ранее данных?)
7. Есть пять полей, уникально идентифицирующих каждую запись в кэше сервера. В каком порядке следует их сравнивать, чтобы минимизировать количество сравнений?
8. При просмотре передаваемых пакетов с помощью `tcpdump` в примере из раздела 16.5, где использовался TCP, мы узнаем, что размер запроса 48 байт, а размер ответа 32 байт (без заголовков TCP и IPv4). Получите этот размер из рисунка 16.3. Каков был бы размер при использовании UDP вместо TCP?
9. Может ли клиент RPC в системе, не поддерживающей потоки, вызвать процедуру сервера, скомпилированную с поддержкой потоков? Что можно сказать о различии в передаваемых аргументах, о котором говорилось в разделе 16.2?
10. В программе `read` из листинга 16.19 мы выделяли место под буфер, в который считывался файл, и этот буфер содержал указатель `vstring_arg`. Но где хранится строка, на которую указывает `vstring_arg`? Измените программу так, чтобы проверить ваше предположение.
11. Sun RPC определяет нулевую процедуру как процедуру с номером 0 (по этой причине мы всегда начинали нумерацию процедур с 1, как в листинге 16.1). Более того, любая заглушка сервера, созданная `grcsen`, автоматически определяет эту процедуру (в чём вы можете легко убедиться, посмотрев текст любой заглушки, созданной для одного из наших примеров). Нулевая процедура не принимает никаких аргументов и ничего не возвращает. Часто она используется для проверки работы сервера и измерения скорости передачи пакетов на сервер и обратно. Но если мы посмотрим на заглушки клиента, мы увидим, что в ней не содержится заглушки для этой процедуры. Посмотрите в документации описание функции `clnt_call` и используйте её для вызова нулевой процедуры для любого сервера этой главы.
12. Почему в табл. A.1 нет записи для сообщения размером 65536 для Sun RPC поверх UDP? Почему нет записей для сообщений длиной 16384 и 32768 в табл. A.2 для Sun RPC поверх UDP?
13. Проверьте, что удаление вызова `xdr_free` из листинга 16.19 приведет к утечке памяти. Добавьте оператор

непосредственно перед вызовом `xdrmem_create` и завершающую скобку непосредственно перед вызовом `xdr_free`. Запустите программу и следите за её размером в памяти с помощью `ps`. Удалите закрывающую скобку и поставьте её после вызова `xdr_free`. Запустите программу снова и последите за её размером еще раз.

Эпилог

В этой книге подробно описаны четыре средства межпроцессного взаимодействия (IPC):

1. Передача сообщений (именованные и неименованные каналы, очереди сообщений Posix и System V).
2. Синхронизация (взаимные исключения и условные переменные, блокировки чтения-записи, блокировки файлов и записей, семафоры Posix и System V).
3. Разделяемая память (неименованная, именованная стандартов Posix и System V).
4. Вызовы процедур (двери в системе Solaris, пакет Sun RPC).

Передача сообщений и вызовов процедур часто используются сами по себе, без дополнительных средств синхронизации. Разделяемая память, напротив, для нормального функционирования обычно требует введения дополнительной синхронизации. Средства синхронизации иногда используются сами по себе, то есть в отдельности от прочих средств IPC.

После прочтения 16 глав возникает естественный вопрос: какую форму IPC следует использовать для решения какой-либо конкретной задачи? К сожалению, универсального метода не существует. Огромное количество средств IPC в Unix возникло благодаря тому, что нет какого-либо единственного средства, которым можно было бы решить все задачи (или хотя бы большинство). Все, что вам остается, — это познакомиться с особенностями всех форм IPC и учитывать их при разработке вашего приложения.

Прежде всего перечислим главные по важности моменты, которые следует учесть при выборе средств организации IPC для приложения.

1. Сетевое или несетевое. Мы предполагаем, что это решение уже принято и IPC используется между процессами или потоками, выполняющимися на одном узле. Если есть вероятность того, что приложение будет распределено между несколькими узлами, следует рассмотреть возможность использования сокетов вместо IPC, чтобы впоследствии легко было переделать приложение в сетевое.
2. Переносимость (вспомните табл. 1.3). Практически все системы под управлением Unix поддерживают именованные и неименованные каналы и блокировку записей стандарта Posix. К 1998 году большинство систем поддерживало средства IPC System V (очереди сообщений, семафоры и разделяемую память), тогда как лишь немногие поддерживали те же средства стандарта Posix. Должны появиться новые реализации Posix IPC, но, к сожалению, эти средства не являются обязательными в стандарте Unix 98. Многие системы поддерживают потоки Posix (включая взаимные исключения и условные переменные) или станут поддерживать их в ближайшем будущем. Некоторые системы, поддерживающие потоки Posix, не воспринимают атрибут использования между процессами для взаимных исключений и условных переменных. Блокировки чтения-записи, требуемые стандартом Unix 98, должны вскоре войти в стандарт Posix, и множество систем уже поддерживают какой-либо из видов таких блокировок. Отображение в память распространено достаточно широко, и большинство Unix-систем поддерживают неименованное отображение (с использованием либо /dev/zero, либо MAP_ANON). Средства Sun RPC должны быть доступны практически на всех системах Unix, тогда как двери пока реализованы только в Solaris.
3. Производительность. Если для вашего приложения критична скорость работы средств IPC, воспользуйтесь программами из приложения А. Лучше всего изменить эти программы для имитации среды, в которой будет работать ваше приложение, и таким образом измерить скорость работы средств IPC в этой среде.
4. Планирование в реальном времени. Если вам нужно воспользоваться этой функцией и система ее поддерживает, рассмотрите возможность использования функций Posix для передачи сообщений и синхронизации (очереди сообщений, семафоры, взаимные исключения и условные переменные). Например, когда увеличиваются значения семафора, в вызове к которому заблокировано несколько потоков, разблокируемый поток выбирается в соответствии с политиками планирования и параметрами заблокированных потоков. Для семафоров System V это не гарантируется.

Чтобы помочь вам понять некоторые особенности и ограничения средств IPC, мы вкратце перечислим основные различия между ними:

- Именованные и неименованные каналы представляют собой потоки байтов без границ между сообщениями. Очереди сообщений Posix и System V предусматривают наличие границ сообщений. Сравните это с протоколами Интернета: TCP — это поток байтов, а UDP — последовательность

сообщений с явно определенными границами.

- Очереди сообщений Posix могут отправлять процессу сигнал или запускать новый поток в случае, если в пустую очередь помещается сообщение. Для очередей System V такая возможность не предусматривается. Ни один из типов очередей сообщений не может быть использован непосредственно с вызовами `select` и `poll` (глава 6 [24]), хотя некие решения этой проблемы были приведены при обсуждении листинга 5.12 и в разделе 6.9.
- Данные в именованных и неименованных каналах передаются в порядке очереди (FIFO). Очереди сообщений Posix и System V предусматривают возможность присваивания сообщениям различного приоритета. При чтении из очереди Posix всегда возвращается сообщение с наивысшим приоритетом. Для очередей System V можно указать любой конкретный тип сообщения.
- При помещении сообщения в очередь Posix или System V или в именованный или неименованный канал ровно один экземпляр доставляется ровно одному считывающему потоку. Возможность передачи нескольким адресатам отсутствует (в отличие от сокетов и XTI при использовании протокола UDP — главы 18 и 19 [24]).
- Взаимные исключения, условные переменные и блокировки чтения-записи имен не имеют. Они могут легко использоваться потоками одного процесса. Совместное использование их различными процессами возможно только в случае, если эти объекты располагаются в общей для этих процессов области памяти. Семафоры Posix бывают двух типов: именованные и размещаемые в памяти. Именованные семафоры могут использоваться только различными процессами (они идентифицируются именами Posix IPC), а размещаемые в памяти должны для этого находиться в разделяемой памяти. Семафоры System V также являются именованными (с помощью типа `key_t`), они также могут без проблем использоваться несколькими процессами совместно.
- Блокировки записей `fcntl` автоматически снимаются ядром при завершении процесса, если он сам об этом не позаботится. Для семафоров System V эта возможность является дополнительной. Для взаимных исключений, условных переменных, блокировок чтения-записи и семафоров Posix эта возможность не предусматривается.
- Каждая блокировка `fcntl` действует на некоторый диапазон байтов (называемый записью) в файле, указываемом с помощью дескриптора. Блокировки чтения-записи не связываются ни с какими записями.
- Разделяемая память Posix и System V обладает живучестью ядра. Она существует до тех пор, пока не будет удалена явно, даже если в какой-то момент не используется ни одним процессом.
- Размер объекта разделяемой памяти Posix может быть увеличен в процессе работы. Размер сегмента разделяемой памяти System V фиксируется при его создании.
- Ограничения ядра на три типа System V IPC часто требуют настройки вручную, поскольку устанавливаемые для них по умолчанию значения часто не соответствуют требованиям реальных приложений (раздел 3.8). Ограничения на средства Posix IPC обычно не требуют настройки.
- Информация об объектах System V IPC (текущий размер, идентификатор владельца, время последнего изменения и т. п.) возвращается вызовом одной из функций `XXXctl` с командой `IPC_STAT` и программой `ipcs`. Для получения информации об объектах Posix стандартных способов не предусматривается. Если реализация использует файлы в качестве основы для этих объектов, можно получить эту информацию с помощью функции `stat` или программы `ls`, если нам известен способ преобразования имени Posix IPC в полное имя файла. Если же в данной реализации файлы не используются, способа получить такую информацию может и не существовать.
- Из всех средств синхронизации — взаимных исключений, условных переменных, блокировок чтения-записи, блокировок записи, семафоров — только две функции можно вызывать из обработчика сигналов (табл. 5.1): `sem_post` и `fcntl`.
- Из всех средств передачи сообщений — каналов, очередей сообщений Posix и System V — только две функции могут быть вызваны из обработчика сигналов: `read` и `write` (используются с именованными и неименованными каналами).
- Из всех средств передачи сообщений только двери предоставляют серверу точную информацию о клиенте (раздел 15.5). В разделе 5.4 мы упомянули два других способа передачи сообщений, которые также предоставляют информацию о клиенте: доменные сокеты в BSD/OS (раздел 14.8 [24]) и каналы в SVR4, если по ним передается дескриптор файла (раздел 15.3.1 [21]).

A.1. Введение

В основной части книги мы перечислили шесть средств передачи сообщений:

- неименованные каналы (pipes);
- именованные каналы (FIFO);
- очереди сообщений Posix;
- очереди сообщений System V;
- двери;
- SunRPC.

Кроме того, мы указали пять типов средств синхронизации:

- взаимные исключения и условные переменные;
- блокировки чтения-записи;
- блокировка записей fcntl;
- семафоры Posix;
- семафоры System V.

В этом приложении мы разработаем набор простых программ для измерения производительности этих видов IPC, чтобы иметь возможность аргументировать свой выбор одного из этих средств для конкретной задачи.

При сравнении средств передачи сообщений нас интересуют два параметра:

1. Полоса пропускания (bandwidth) — скорость передачи данных по каналу IPC. Для измерения этого параметра мы передаем огромное количество данных (миллионы байтов) от одного процесса другому. Этот параметр измеряется для различных объемов данных на операцию (например, write и read для каналов), и мы ожидаем, что полоса пропускания будет увеличиваться вместе с увеличением количества передаваемых за одну операцию данных.
2. Задержка (latency) определяется как время, которое требуется небольшому сообщению, чтобы проделать путь по каналу IPC от одного процесса к другому и обратно. Мы измеряем время задержки для сообщения размером 1 байт.

В реальности величина полосы пропускания говорит нам о том, сколько времени будет потрачено на передачу блока данных по каналу IPC, но IPC также используется и для передачи небольших управляющих сообщений. Задержка определяет время, необходимое для передачи этих сообщений. Важными оказываются обе величины.

Чтобы измерить скорость работы средств синхронизации, мы изменим программу, увеличивающую значение счетчика в разделяемой памяти. Поскольку увеличение — элементарная операция, время будет тратиться в основном на работу средств синхронизации.

ПРИМЕЧАНИЕ

Программы этого приложения, используемые для измерения производительности средств IPC, основаны на пакете lmbench [15]. Этот пакет содержит набор тестов, измеряющих множество параметров системы (скорость переключения контекста и т. п.), а не только производительность средств IPC. Исходный код пакета доступен по адресу <http://www.bitmover.com/lmbench>.

Величины, приведенные в этом приложении, дают нам возможность сравнить методы, описанные в книге. Заодно мы хотели показать, как просто измерить эти величины. Прежде чем останавливать свой выбор на одном из средств IPC, нeliшне было бы получить эти значения в вашей собственной системе. К сожалению, насколько легко измерить величины, настолько же трудно объяснить аномальное их поведение в отсутствие доступа к исходному коду ядра и библиотек.

Сведем вместе результаты, полученные в этом приложении. Данный раздел может использоваться как справочник при чтении книги.

Для проведения измерений использовались две системы: SparcStation 4/110 под управлением Solaris 2.6 и Digital Alpha (DEC 3000 model 300, Pelican) под управлением Digital Unix 4.0B. В файл /etc/system системы Solaris 2.6 были добавлены следующие строки:

Это дает возможность отправлять сообщения размером 16384 байт в очередь сообщений System V (табл. А.2). Те же изменения осуществляются в Digital Unix 4.0B введением следующих строк с помощью программы sysconfig:

Результаты измерения полосы пропускания сообщений

В табл. А.2 приведены результаты измерений на компьютере Sparc под управлением Solaris 2.6, а на рис. А.1 — график этих результатов.

Как мы и предполагали, полоса пропускания увеличивается с размером сообщения. Поскольку во многих реализациях очередей сообщений System V ограничение на размер сообщения, установленное в ядре, достаточно мало (раздел 3.8), максимальный размер сообщения в нашей программе имеет значение 16384 байт.

Уменьшение полосы для сообщений размером около 4096 байт в Solaris 2.6, возможно, связано с настройкой внутренних ограничений ядра. Для сравнения с [24] мы приводим результаты аналогичных измерений для сокета TCP и доменного сокета Unix. Эти две величины были получены с помощью программ пакета lmbench для сообщений размером 65536 байт. При измерении быстродействия сокета TCP оба процесса выполнялись на одном узле.

Результаты измерения задержки

В табл. А.1 приведены значения задержки в Solaris 2.6 и Digital Unix 4.0B.

Таблица А.1. Задержка при передаче сообщения размером 1 байт (в микросекундах)



Рис. А.1. Полоса пропускания средств передачи сообщений в Solaris 2.6.

В разделе А.4 мы приведем листинги программ, использованных для получения первых четырех величин, а оставшиеся три получены с помощью пакета lmbench. При измерении скорости работы TCP и UDP оба процесса находились на одном узле.

Таблица А.2. Полоса пропускания для разных типов сообщений в Solaris 2.6 (Мбайт/с)



Рис. А.2. Полоса пропускания для различных средств передачи сообщений (Digital Unix 4.0B)

Таблица А.3. Полоса пропускания для различных типов сообщения в Digital Unix 4.0B (Мбайт/с)

Результаты синхронизации потоков

В табл. А.4 приведены значения времени, нужного одному или нескольким потокам для увеличения счетчика в разделяемой памяти с использованием различных средств синхронизации в Solaris 2.6, а на рис. А.3 показан график этих значений. Каждый поток увеличивает значение счетчика 1000000 раз, а количество потоков меняется от 1 до 5. В табл. А.5 приведены эти же значения для Digital Unix 4.0B, а на рис. А.4 — график этих значений.



Рис. А.3. Время увеличения счетчика в разделяемой памяти (Solaris 2.6)

Мы увеличиваем количество потоков, чтобы проверить правильность кода. Кроме того, при добавлении потоков время работы программы может начать расти нелинейно. Блокировка fcntl может использоваться только одним потоком, поскольку эта форма синхронизации предназначена только для использования между несколькими процессами, а не потоками одного процесса.

В Digital Unix 4.0B значения для семафоров Posix оказываются непомерно большими, если работает более одного потока, что указывает на наличие какой-то аномалии. На графике мы эти значения не приводим.

ПРИМЕЧАНИЕ

Одна из возможных причин этой аномалии заключается в том, что в этой программе синхронизация проверяется неправильно. В данном случае потоки не делают ничего полезного, и все время уходит на синхронизацию. Поскольку потоки создаются с внутрипроцессной конкуренцией, при потере управления потоком блокировка сохраняется, поэтому поток, получающий управление, выполняться дальше не может.



Рис.А.4. Время увеличения счетчика в разделяемой памяти (Digital Unix 4.0B)

Таблица А.4. Время увеличения счетчика в разделяемой памяти для Solaris 2.6 (в секундах)

Таблица А.5. Время увеличения счетчика в разделяемой памяти в Digital Unix 4.0B (в секундах)

Результаты синхронизации процессов

В табл. А.4 и А.5 и на соответствующих рисунках были приведены результаты синхронизации потоков одного процесса. Интересно посмотреть, как взаимодействуют разные процессы. В табл. А.6 и на рис. А.5 приведены результаты измерения времени увеличения счетчика несколькими процессами в Solaris 2.6, а в табл. А.7 и на рис. А.6 — в Digital Unix 4.0B. Результаты похожи на полученные для потоков, однако в Solaris 2.6 теперь получаются одинаковые результаты для первых двух типов семафоров. Мы приводим на графике только первое значение для fcntl, поскольку последующие слишком велики. Как отмечалось в разделе 7.2, Digital Unix 4.0B не поддерживает атрибут PTHREAD_PROCESS_SHARED, поэтому мы не можем измерить скорость работы взаимных исключений в этой системе. Для семафоров Posix в Digital Unix 4.0B опять наблюдаются аномалии.



Рис. А.5. Время увеличения счетчика в разделяемой памяти (Solaris 2.6)

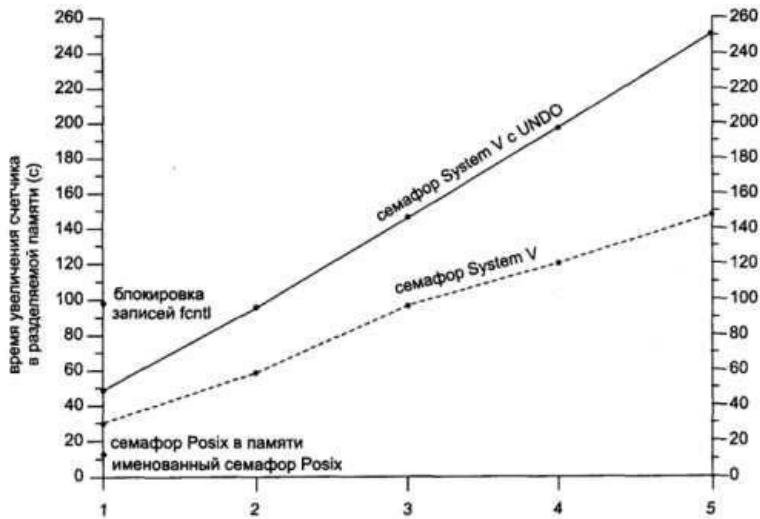


Рис. А.6. Время увеличения счетчика в разделяемой памяти

Таблица А.6. Время увеличения счетчика в разделяемой памяти для Solaris 2.6 (в секундах)

Таблица А.7. Время увеличения счетчика в разделяемой памяти для Digital Unix 4.0B (в секундах)

В этом разделе приведены тексты трех программ, измеряющих полосу пропускания каналов, очередей сообщений Posix и System V. Результаты работы этих программ приведены в табл. А.2 и А.3.

Измерение полосы пропускания канала

На рис. А.7 приведена схема описываемой программы.



Рис. А.7. Схема программы измерения полосы пропускания канала

В листинге А.1 приведен текст первой половины программы bw_pipe, измеряющей полосу пропускания канала.

11-15 Аргументы командной строки задают количество повторов (обычно 5), количество передаваемых мегабайтов (если указать 10, будет передано $10 \times 1024 \times 1024$ байт) и количество байтов для каждой операции read и write (которое может принимать значения от 1024 до 65536 в наших измерениях).

16-17 Вызов valloc аналогичен malloc, но выделяемая память начинается с границы страницы памяти. Функция touch (листинг А.3) помещает 1 байт данных в каждую страницу буфера, заставляя ядро считать в память все страницы данного буфера. Мы всегда выполняем это перед проведением измерений.

ПРИМЕЧАНИЕ

Функция valloc не входит в стандарт Posix.1 и названа устаревшей в Unix 98. Она требовалась в ранних версиях спецификаций X/Open, но уже не является необходимой. Обертка Valloc вызывает функцию malloc, если valloc недоступна.

18-19 Создаются два канала: contpipe[0] и contpipe[1] используются для синхронизации процессов перед началом передачи, а datapipe[0] и datapipe[1] используются для передачи самих данных.

20-31 Создается дочерний процесс, вызывающий функцию writer, а родительский процесс в это время вызывает функцию reader. Функция reader вызывается nloop раз. Функция start_time вызывается непосредственно перед началом цикла, а stop_time — сразу после его окончания. Эти функции даны в листинге А.3. Полоса пропускания представляет собой количество байтов, переданных за все проходы цикла, поделенное на время, затраченное на передачу (stop_time возвращает количество микросекунд, прошедшее с момент запуска start_time). Затем дочерний процесс завершается сигналом SIGTERM и программа завершает свою работу. Вторая половина программы приведена в листинге А.2. Она состоит из функций reader и writer.

33-44 Функция writer представляет собой бесконечный цикл, вызываемый дочерним процессом. Он ожидает сообщения родительского процесса о готовности к приему данных, считывая целое число из управляющего канала. Это целое число определяет количество байтов, которое будет записано в канал данных. При получении этого числа дочерний процесс записывает данные в канал, отправляя их родителю. За один вызов write записывается xfersize байтов.

45-54 Эта функция вызывается родительским процессом в цикле. Каждый раз при вызове функции в управляющий канал записывается целое число, указывающее дочернему процессу на необходимость помещения соответствующего количества данных в канал данных. Затем функция вызывает read в цикле до тех пор, пока не будут приняты все данные.

Текст функций start_time, stop_time и touch приведен в листинге А.3.

Текст функции tv_sub приведен в листинге А.4. Она осуществляет вычитание двух структур timeval, сохраняя результат в первой структуре.

На компьютере Sparc под управлением Solaris 2.6 при выполнении программы пять раз подряд получим следующий результат:

Каждый раз мы задаем пять циклов, 10 Мбайт за цикл и 65536 байт за один вызов write или read. Среднее от этих пяти результатов даст величину 13,7 Мбайт в секунду, приведенную в табл. А.2.

Измерение полосы пропускания очереди сообщений Posix

В листинге A.5 приведена функция main программы, измеряющей полосу пропускания очереди сообщений Posix. Листинг A.6 содержит функции reader и writer. Эта программа устроена аналогично предыдущей, измерявшей полосу пропускания канала.

ПРИМЕЧАНИЕ

Обратите внимание, что в программе приходится указывать максимальное количество сообщений в очереди при ее создании. Мы указываем значение 4. Размер канала IPC может влиять на производительность, потому что записывающий процесс может отправить это количество сообщений, прежде чем будет заблокирован в вызове mq_send, что приведет к переключению контекста на считывающий процесс. Следовательно, производительность программы зависит от этого магического числа. Изменение его с 4 на 8 в Solaris 2.6 никак не влияет на величины, приведенные в табл. А.2, но в Digital Unix 4.0B производительность уменьшается на 12%. Мы могли ожидать, что производительность возрастет с увеличением количества сообщений в очереди, поскольку требуется в два раза меньше переключений контекста. Однако если используется отображение файла в память, это увеличивает размер отображаемого файла в два раза, как и требуемое количество памяти.

Программа измерения полосы пропускания очереди System V

В листинге A.7 приведен текст функции main, измеряющей полосу пропускания очередей сообщений System V, а в листинге A.8 —текст функций reader и writer.

Программа измерения полосы пропускания дверей

Программа измерения полосы пропускания интерфейса дверей сложнее, чем предыдущие, поскольку нам нужно вызвать fork перед созданием двери. Родительский процесс создает дверь и с помощью канала оповещает дочерний процесс о том, что ее можно открывать.

Другое изменение заключается в том, что в отличие от рис. A.7 функция reader не принимает данные. Данные принимаются функцией server, которая является процедурой сервера для данной двери. На рис. A.8 изображена схема программы.



Рис. А.8. Схема программы измерения полосы пропускания дверей

Поскольку двери поддерживаются только в Solaris, мы упростим программу, предполагая наличие двустороннего канала (раздел 4.4).

Еще одно изменение вызвано фундаментальным различием между передачей сообщений и вызовом процедуры. В программе, работавшей с очередью сообщений Posix, например, записывающий процесс просто помещал сообщения в очередь в цикле, что осуществляется асинхронно. В какой-то момент очередь будет заполнена или записывающий процесс будет просто приостановлен, и тогдачитывающий процесс получит сообщения. Если, например, в очередь помещается 8 сообщений и записывающий процесс помещал в нее 8 сообщений каждый раз, когда получал управление, а считающий процесс считывал 8 сообщений, отправка N сообщений требовала $N/4$ переключения контекста. Интерфейс дверей является синхронным:зывающий процесс блокируется каждый раз при вызове door call и не может возобновиться до тех пор, пока сервер не завершит работу. Передача N сообщений в этом случае требует $N/2$ переключений контекста. С той же проблемой мы столкнемся при измерении полосы пропускания вызовов RPC. Несмотря на увеличившееся количество переключений контекста, из рис. А.1 следует, что двери обладают наибольшей полосой пропускания при размере сообщений не более 25 Кбайт.

В листинге А.9 приведен текст функции main нашей программы. Функции writer, server и reader приведены в листинге А.10.

Программа определения полосы пропускания Sun RPC

Поскольку вызовы процедур в Sun RPC являются синхронными, для них действует то же ограничение, что и для дверей (см. выше). В данном случае проще создать две программы (клиент и сервер), поскольку они создаются автоматически программой grcsen. В листинге A.11 приведен файл спецификации RPC. Мы объявляем единственную процедуру, принимающую скрытые данные переменной длины в качестве входного аргумента и ничего не возвращающую.

В листинге A.12 приведен текст программы-клиента, а в листинге A.13 — процедура сервера. Мы указываем протокол в качестве аргумента командной строки при вызове клиента, что позволяет нам измерить скорость работы обоих протоколов.

Приведем текст трех программ, измеряющих задержку при передаче сообщений по каналам, очередям Posix и очередям System V. Данные о производительности, полученные с их помощью, приведены в табл. А.1.

Программа измерения задержки канала

Программа для измерения задержки канала приведена в листинге А.14.

2-9 Эта функция запускается родительским процессом. Мы измеряем время ее работы. Она помещает 1 байт в канал, из которого читает дочерний процесс, и считывает 1 байт из другого канала, в который сообщение помещается дочерним процессом. При этом измеряется именно то, что мы назвали задержкой, — время передачи небольшого сообщения туда и обратно.

19-20 Создаются два канала, после чего вызов fork порождает дочерний процесс. При этом образуется схема, изображенная на рис. 4.6 (но без закрытия неиспользуемых дескрипторов каналов). Для этого теста требуются два канала, поскольку каналы являются односторонними, а мы хотим передавать сообщение в обе стороны.

22-27 Дочерний процесс представляет собой бесконечный цикл, в котором однобайтовое сообщение считывается и отсылается обратно.

29-34 Родительский процесс вызывает функцию doit для отправки однобайтового сообщения дочернему процессу и получения ответа. После этого мы имеем гарантию, что оба процесса выполняются. Затем функция doit вызывается в цикле с измерением времени задержки.

На компьютере Sparc под управлением Solaris 2.6 при запуске программы пять раз подряд мы получим вот что:

Среднее для пяти попыток составляет 324 микросекунды, и именно это значение приведено в табл. А.1. Это время учитывает два переключения контекста (от родительского процесса к дочернему и обратно), четыре системных вызова (write, read, write, read) и затраты на передачу 1 байта данных по каналу.

Программа измерения задержки очередей сообщений Posix

Программа измерения задержки для очередей сообщений Posix приведена в листинге A.15.

25-28 Создаются две очереди сообщений, каждая из которых используется для передачи данных в одну сторону. Хотя для очередей Posix можно указывать приоритет сообщений, функция `mq_receive` всегда возвращает сообщение с наивысшим приоритетом, поэтому мы не можем использовать лишь одну очередь для данного приложения.

Измерение задержки очередей сообщений System V

В листинге A.16 приведен текст программы измерения времени задержки для очередей сообщений System V.

Мы создаем одну очередь, по которой сообщения передаются в обоих направлениях. Сообщения с типом 1 передаются от родительского процесса дочернему, а сообщения с типом 2 — в обратную сторону. Четвертый аргумент при вызове `msgrecv` в функции `doit` имеет значение 2, что обеспечивает получение сообщений только данного типа. Аналогично в дочернем процессе четвертый аргумент `msgrecv` имеет значение 1.

ПРИМЕЧАНИЕ

В разделах 9.3 и 11.3 мы отмечали, что многие структуры, определенные в ядре, нельзя инициализировать статически, поскольку стандарты Posix.1 и Unix 98 гарантируют лишь наличие определенных полей в этих структурах, но не определяют ни их порядок, ни наличие других полей. В этой программе мы инициализируем структуру `msgbuf` статически, поскольку очереди сообщений System V гарантируют, что эта структура содержит поле типа сообщения `long`, за которым следуют передаваемые данные.

Программа измерения задержки интерфейса дверей

Программа измерения задержки для интерфейса дверей дана в листинге А.17. Дочерний процесс создает дверь и связывает с ней функцию server. Родительский процесс открывает дверь и вызывает door_call в цикле. В качестве аргумента передается 1 байт данных, и ничего не возвращается.

Программа измерения времени задержки Sun RPC

Для измерения времени задержки Sun RPC мы напишем две программы: клиент и сервер, аналогично измерению полосы пропускания. Мы используем старый файл спецификации RPC, но на этот раз клиент вызывает нулевую процедуру сервера. Вспомните упражнение 16.11: эта процедура не принимает никаких аргументов и ничего не возвращает. Это именно то, что нам нужно, чтобы определить задержку. В листинге A.18 приведен текст клиента. Как и в решении упражнения 16.11, нам нужно воспользоваться `clnt_call` для вызова нулевой процедуры; в заглушке клиента отсутствует необходимая заглушка для этой процедуры.

Мы компилируем сервер с функцией, приведенной в листинге A.13, но она все равно не вызывается. Поскольку мы используем `grcsdgen` для построения клиента и сервера, нам нужно определить хотя бы одну процедуру сервера, но мы не обязаны ее вызывать. Причина, по которой мы используем `grcsdgen`, заключается в том, что она автоматически создает функцию `main` сервера с нулевой процедурой, которая нам нужна.

Для измерения времени, уходящего на синхронизацию при использовании различных средств, мы создаем некоторое количество потоков (от одного до пяти, согласно табл. А.4 и А.5), каждый из которых увеличивает счетчик в разделяемой памяти большое количество раз, используя различные формы синхронизации для получения доступа к счетчику.

Взаимные исключения Posix

В листинге А.19 приведены глобальные переменные и функция main программы, измеряющей быстродействие взаимных исключений Posix.

4-9 Совместно используемые потоками данные состоят из взаимного исключения и счетчика. Взаимное исключение инициализируется статически.

20-26 Основной поток блокирует взаимное исключение перед созданием прочих потоков, чтобы ни один из них не получил это исключение до тех пор, пока все они не будут созданы. Вызывается функция set_concurrency, создаются потоки. Каждый поток выполняет функцию incr, текст которой будет приведен позже.

27-36 После создания всех потоков главный поток запускает таймер и освобождает взаимное исключение. Затем он ожидает завершения всех потоков, после чего останавливает таймер и выводит полное время работы. В листинге А.20 приведен текст функции incr, выполняемой каждым из потоков.

44-46 Операция увеличения счетчика осуществляется после получения блокировки на взаимное исключение. После этого взаимное исключение разблокируется.

Блокировки чтения-записи

Программа, использующая блокировки чтения-записи, является слегка измененной версией программы с взаимными исключениями Posix. Поток должен установить блокировку файла, прежде чем увеличивать общий счетчик.

ПРИМЕЧАНИЕ

Существует не так уж много систем, в которых реализованы блокировки чтения-записи, являющиеся частью стандарта Unix 98 и разрабатываемые рабочей группой Posix.1j. Измерения в этом разделе проводились в системе Solaris 2.6 с использованием блокировок, описанных в документации на странице [rwlock\(3T\)](#). Эта реализация обеспечивает тот же набор функций, что и предлагаемые блокировки чтения-записи Posix. Для использования этих функций мы применяем тривиальные функции-обертки.

В Digital Unix 4.0B мы использовали блокировки чтения-записи поточно-независимых служб, описанные на странице документации [tis_rwlock](#). Мы не приводим листингов с несущественными изменениями, необходимыми для использования этих блокировок.

В листинге A.21 приведен текст функции main, а в листинге A.22 — текст функции incr.

Семафоры Posix, размещаемые в памяти

Мы измеряем скорость работы семафоров Posix (именованных и размещаемых в памяти). В листинге A.24 приведен текст функции main, а в листинге A.23 — текст функции incr.

18-19 Создается семафор, инициализируемый значением 0. Второй аргумент в вызове sem_init, имеющий значение 0, говорит о том, что семафор используется только потоками вызвавшего процесса.

20-27 После создания всех потоков запускается таймер и вызывается функция sem_post.

Именованные семафоры Posix

В листинге A.26 приведен текст функции main, измеряющей быстродействие именованных семафоров Posix, а в листинге A.25 — соответствующая функция incr.

Семафоры System V

Функция main программы, измеряющей быстродействие семафоров System V, приведена в листинге A.27, а функция incr показана в листинге A.28.

20-23 Создается семафор с одним элементом, значение которого инициализируется нулем.

24-29 Инициализируются две структуры semop: одна для увеличения семафора, а другая для ожидания его изменения. Обратите внимание, что поле sem_flg в обеих структурах имеет значение 0: флаг SEM_UNDO не установлен.

Единственное отличие от программы из листинга A.27 заключается в том, что поле sem_flg структур semop устанавливается равным SEM_UNDO, а не 0. Мы не приводим в книге новый листинг с этим небольшим изменением.

Блокировка записей fcntl

Последняя программа использует fcntl для синхронизации. Функция main приведена в листинге A.30. Эта программа будет выполняться успешно только в том случае, если количество потоков равно 1, поскольку блокировка fcntl предназначена для использования между процессами, а не между потоками одного процесса. При указании нескольких потоков каждый из них всегда имеет возможность получить блокировку (то есть вызовы write_lock не приводят к остановке потока, потому что процесс уже является владельцем блокировки), и конечное значение счетчика оказывается неправильным.

Функция incr, использующая блокировку записей, приведена в листинге A.29.

15-19 Полное имя создаваемого и используемого для блокировки файла принимается в качестве аргумента командной строки. Это позволяет измерять скорость работы для разных файловых систем. Можно ожидать, что программа будет работать гораздо медленнее при использовании NFS (если она вообще будет работать, то есть если сервер и клиент NFS поддерживают блокировку записей NFS).

В программах предыдущего раздела счетчик использовался несколькими потоками одного процесса. При этом он представлял собой глобальную переменную. Теперь нам нужно изменить эти программы для измерения скорости синхронизации различных процессов.

Для разделения счетчика между родительским процессом и дочерними мы помещаем его в разделяемую память, выделяемую функцией `my_shm`, показанной в листинге A.31.

Если система поддерживает флаг `MAP_ANON` (раздел 12.4), мы используем этот тип разделяемой памяти. В противном случае используется отображение в память файла `/dev/zero` (раздел 12.5).

Последующие изменения зависят от средства синхронизации и от того, что происходит с лежащим в основе этого средства типом данных при вызове `fork`. Детали описаны в разделе 10.12.

- Взаимное исключение Posix: должно храниться в разделяемой памяти (вместе со счетчиком) и инициализироваться с установленным атрибутом `PTHREAD_PROCESS_SHARED`. Код программы будет приведен ниже.
- Блокировка чтения-записи Posix: должна храниться в разделяемой памяти (вместе со счетчиком) и инициализироваться с установленным атрибутом `PTHREAD_PROCESS_SHARED`.
- Семафоры Posix, размещаемые в памяти: семафор должен храниться в разделяемой памяти (вместе со счетчиком), и вторым аргументом при вызове `sem_init` должна быть единица (указывающая на то, что семафор используется несколькими процессами).
- Именованные семафоры Posix: следует либо вызывать `sem_open` из родительского и дочерних процессов по отдельности, либо вызывать `sem_open` в родительском процессе, учитывая, что семафор станет общим после вызова `fork`.
- Семафоры System V: никакого специального кодирования не требуется, поскольку эти семафоры всегда могут использоваться как процессами, так и потоками. Дочерним процессам достаточно знать идентификатор семафора.
- Блокировка записей `fcntl`: изначально предназначена для использования несколькими процессами.

Мы приведем код только для программы с взаимными исключениями Posix.

Взаимные исключения Posix между процессами

Функция main первой программы использует взаимное исключение Posix для обеспечения синхронизации. Текст ее приведен в листинге A.32.

19-20 Поскольку мы запускаем несколько процессов, структура shared должна располагаться в разделяемой памяти. Мы вызываем функцию my_shm, текст которой приведен в листинге A.31.

21-26 Поскольку взаимное исключение помещено в разделяемую память, мы не можем статически инициализировать его, поэтому мы вызываем pthread_mutex_init после установки атрибута PTHREAD_PROCESS_SHARED. Взаимное исключение блокируется.

27-36 После создания дочерних процессов и запуска таймера блокировка снимается.

37-43 Родительский процесс ожидает завершения всех дочерних, после чего останавливает таймер.

Б.1. Введение

В этом приложении приведены основные функции, используемые для работы с потоками. В традиционной модели Unix процесс, которому нужно, чтобы какое-то действие было выполнено не им самим, порождает дочерний процесс вызовом fork. Большая часть сетевых серверов под Unix написана именно так.

Хотя эта парадигма хорошо работала на протяжении многих лет, вызов fork обладает некоторыми недостатками:

- вызов fork ресурсоемок. Память копируется от родительского процесса к дочернему, копируются все дескрипторы и т. д. Существующие реализации используют метод копирования при записи (copy-on-write), что исключает необходимость копирования адресного пространства родительского процесса, пока оно не понадобится клиенту, но, несмотря на эту оптимизацию, вызов fork остается ресурсоемким;
- для передачи информации между родительским и дочерним процессами необходимо использовать одну из форм IPC после вызова fork. Передать информацию дочернему процессу легко: это можно сделать до вызова fork. Однако передать ее обратно может быть достаточно сложно.

Потоки помогают решить обе проблемы. Часто они называются «облегченными процессами» (lightweight processes), поскольку поток проще, чем процесс. Создание потока может занимать по времени меньше одной десятой создания процесса.

Все потоки одного процесса совместно используют его глобальные переменные, поэтому им легко обмениваться информацией, но это приводит к необходимости синхронизации. Однако общими становятся не только глобальные переменные. Все потоки одного процесса разделяют:

- инструкции процесса;
- большую часть данных;
- открытые файлы (дескрипторы);
- обработчики сигналов и вообще настройки для работы с сигналами;
- текущий рабочий каталог;
- идентификатор пользователя и группы.

Однако каждый поток имеет свои собственные:

- идентификатор потока;
- набор регистров, включая PC и указатель стека;
- стек (для локальных переменных и адресов возврата);
- errno;
- маску сигналов;
- приоритет.

Б.2. Основные функции для работы с потоками: создание и завершение

В этом разделе мы опишем пять основных функций для работы с потоками.

При запуске программы вызовом exec создается единственный поток, называемый начальным потоком, или главным (initial thread). Добавочные потоки создаются вызовом pthread_create:

Каждый поток процесса обладает собственным идентификатором потока, который имеет тип pthread_t. При успешном создании нового потока его идентификатор возвращается через указатель *tid*.

Каждый поток обладает некоторым количеством атрибутов: приоритетом, начальным размером стека, признаком демона и т. п. При создании потока эти атрибуты могут быть указаны с помощью переменной типа pthread_attr_t, значение которой имеет более высокий приоритет, чем значения по умолчанию. Обычно мы используем значения по умолчанию. При этом аргумент attr является нулевым указателем.

Наконец, при создании потока мы должны указать функцию, которую он будет выполнять, — начальную функцию потока (thread start function). Поток запускается вызовом этой функции и завершается либо явно (вызовом pthread_exit), либо неявно (возвратом из этой функции). Адрес функции указывается в аргументе *func*, и вызывается она с единственным аргументом — указателем *arg*. Если функции нужно передать несколько аргументов, следует упаковать их в структуру и передать ее адрес в качестве единственного аргумента начальной функции.

Обратите внимание на объявления *func* и *arg*. Функция принимает один аргумент — указатель типа void, и возвращает один аргумент — такой же указатель. Это дает нам возможность передать потоку указатель на что угодно и получить в ответ такой же указатель.

Функции Posix для работы с потоками обычно возвращают 0 в случае успешного завершения работы и ненулевое значение в случае ошибки. В отличие от большинства системных функций, возвращающих -1 в случае ошибки и устанавливающих значение errno равным коду ошибки, функции Pthread возвращают положительный код ошибки. Например, если pthread_create не сможет создать новый поток из-за превышения системного ограничения на потоки, эта функция вернет значение EAGAIN. Функции Pthread не устанавливают значение переменной errno. Несоответствие при их вызове не возникает, поскольку ни один из кодов ошибок не имеет нулевого значения (<sys/errno.h>).

Мы можем ожидать завершения какого-либо процесса, вызвав pthread_join. Сравнивая потоки с процессами Unix, можно сказать, что pthread_create аналогична fork, а pthread_join — waitpid:

Мы должны указать идентификатор потока, завершения которого ожидаем. К сожалению, невозможно задать режим ожидания завершения нескольких потоков (аналога waitpid с идентификатором процесса -1 нет).

Если указатель *status* ненулевой, возвращаемое потоком значение (указатель на объект) сохраняется в ячейке памяти, на которую указывает *status*.

У каждого потока имеется свой идентификатор, уникальный в пределах данного процесса. Идентификатор возвращается pthread_create и используется при вызове pthread_join. Поток может узнать свой собственный идентификатор вызовом pthread_self:

Вызов pthread_self является аналогом getpid для процессов Unix.

Поток может являться как присоединяемым (по умолчанию), так и отсоединенным. При завершении присоединяемого потока его идентификатор и статус завершения сохраняются до тех пор, пока какой-либо другой поток данного процесса не вызовет pthread_join. Отсоединенный поток функционирует аналогично процессу-демону. После его завершения все ресурсы освобождаются. Никакой другой поток не может ожидать его завершения. Если имеется необходимость ожидания одним потоком завершения другого, лучше оставить последний присоединяемым.

Функция pthread_detach делает данный поток отсоединенным:

Эта функция вызывается потоком при необходимости изменить собственный статус в форме

Одним из способов завершения потока является вызов pthread_exit:

Если поток не является отсоединенным, его идентификатор и статус завершения сохраняются для возвращения другому потоку, который может вызвать pthread_join.

Указатель *status* не должен быть установлен на локальный объект вызвавшего потока (типа автоматической переменной), поскольку этот объект уничтожается при завершении потока.

Поток может быть завершен двумя другими способами:

- начальная функция потока (третий аргумент `pthread_create`) может вызвать `return`. Поскольку эта функция должна объявляться как возвращающая указатель на тип `void`, это возвращаемое значение становится статусом завершения потока;
- функция `main` процесса может завершить работу или один из потоков может вызвать `exit` или `_exit`. При этом процесс завершает работу немедленно, вместе со всеми своими потоками.

B.1. Заголовочный файл upripc.h

Почти все программы книги подключают заголовочный файл upripc.h, приведенный в листинге B.1. Он подключает все стандартные системные заголовки, нужные большинству программ для работы с сетью, вместе с некоторыми общими системными заголовками. Он также определяет константы типа MAXLINE и прототипы функций ANSI C для функций, определенных в тексте (типа px_ipc_name), и для всех используемых в книге оберток. Мы не приводим эти прототипы.

B.2. Заголовочный файл config.h

Для подготовки программ в этой книге использовалась утилита GNU autoconf, которая помогает сделать их более переносимыми. Она доступна по адресу <ftp://prep.ai.mit.edu/pub-gnu>. Утилита создает сценарий configure, который следует запустить после того, как вы загрузите программу из Сети. Этот сценарий определяет возможности вашей системы: поддерживаются ли очереди System V, определен ли тип uint8_t, определена ли функция gethostname и т. д. В процессе работы он создает заголовочный файл config.h, который включается нашим iprisc.h перед всеми остальными. В листинге B.2 приведен пример заголовочного файла config.h для системы Solaris 2.6 и компилятора gcc.

Строки, начинающиеся с #define, описывают функции, поддерживаемые в системе. Закомментированные строки соответствуют неподдерживаемым функциям.

В.3. Стандартные функции вывода сообщений об ошибках

Мы определили свой набор функций, используемых во всех программах книги для обработки ситуаций с возникновением ошибок. Причина, по которой мы создаем эти функции, заключается в том, что теперь мы можем писать команды в одну строку:

вместо:

Функции обработки ошибок используют возможности работы со списком аргументов переменной длины, определенные стандартом ANSI C. В разделе 7.3 [11] вы можете узнать подробности.

В таблице В.1 приведены отличия между различными функциями обработки ошибок. Если глобальное целое daemon_proc отлично от нуля, сообщение передается демону syslog с указанным уровнем (см. главу 12 [24]); в противном случае сообщение выводится в стандартный поток сообщений об ошибках.

Таблица В.1. Функции обработки ошибок

В листинге В.3 приведен текст функций из табл. В.1.

Глава 1

1. В обоих процессах нужно лишь указать флаг O_APPEND при вызове функции open или режим дополнения файла при вызове fopen. Ядро гарантирует, что данные будут дописываться в конец файла. Это самая простая форма синхронизации доступа к файлу. На с. 60-61 [21] об этом рассказывается более подробно. Синхронизация становится проблемой при обновлении имеющихся в файле данных, как это происходит в базах данных.

2. Обычно встречается что-нибудь вроде:

Если определена константа _REENTRANT, обращение к errno приводит к вызову функции _errno, возвращающей адрес переменной errno вызвавшего потока. Эта переменная, скорее всего, хранится в области собственных данных этого потока (раздел 23.5 [24]). Если константа REENTRANT не определена, переменная errno является глобальной.

Глава 2

1. Эти два бита могут менять действующий идентификатор пользователя и/или группы выполняющейся программы. Идентификаторы используются в разделе 2.4.
2. Сначала следует указать флаги O_CREAT | O_EXCL, и если вызов окажется успешным, будет создан новый объект. Если вызов вернет ошибку EEXIST, объект уже существует и программа должна вызвать open еще раз, без флага O_CREAT или O_EXCL Второй вызов должен оказаться успешным, но есть вероятность, что он вернет ошибку ENOENT, если какой-либо другой поток или процесс удалит объект в промежутке между этими двумя вызовами.

Глава 3

1. Текст программы приведен в листинге Г.1.

2. Первый вызов `msgget` задействует первую свободную очередь сообщений, порядковый номер которой имеет значение 20 после двукратного запуска программы из листинга 3.2, и вернет идентификатор 1000. Если предположить, что следующая доступная очередь сообщений никогда ранее не использовалась, ее порядковый номер будет иметь значение 0, а возвращаться будет идентификатор 1.

3. Программа приведена в листинге Г.2.

Запустив эту программу, мы увидим, что маска создания файла имеет значение 2 (снять бит записи для прочих пользователей) и этот бит оказывается снятым для канала FIFO, но не для очереди сообщений:

4. При использовании `ftok` имеется вероятность того, что для двух полных имен получится один и тот же ключ. При использовании `IPC_PRIVATE` сервер знает, что он создает новую очередь, но в этом случае ему нужно записать ее идентификатор в какой-либо файл, чтобы клиенты могли его считать.

5. Вот один из способов обнаружения коллизий:

Программа `find` игнорирует файлы, на которые имеется несколько ссылок (поскольку у всех ссылок одинаковый номер узла), и символические ссылки (поскольку функция `stat` возвращает информацию для файла, на который ссылка указывает). Большой процент коллизий (75,2%) вызван тем, что в Solaris 2.x используется только 12 бит номера узла. Поэтому в файловых системах с числом файлов более 4096 количество коллизий может быть велико. Например, файлы с номерами 4096, 8192, 12288 и 16384 будут иметь один и тот же ключ `IPC` (если все они принадлежат одной файловой системе).

Мы запустили эту программу в той же файловой системе, но используя функцию `ftok` из BSD/OS, которая добавляет номер узла к ключу целиком, и получили всего 849 коллизий (менее 1%).

Глава 4

1. Если бы дескриптор fd[1] остался открытым в дочернем процессе при завершении родительского, его операция read для этого дескриптора не вернула бы признак конца файла, потому что дескриптор был бы еще открыт в дочернем процессе. Закрытие fd[1] гарантирует, что после завершения родительского процесса все его дескрипторы закрываются и вызов read для fd[1] возвращает 0.

2. Если поменять местами порядок вызовов, другой процесс сможет создать канал FIFO в промежутке между вызовами open и mkfifo, в результате чего последний вернет ошибку.

3. Если выполнить

мы увидим, что popen срабатывает успешно, но fgets считывает символ конца файла. Сообщение об ошибке записывается интерпретатором в стандартный поток сообщений об ошибках.

5. Измените первый вызов open, указав флаг отключения блокировки:

Возврат из этого вызова произойдет немедленно, как и из следующего вызова open, поскольку канал уже открыт на чтение. Чтобы избежать ошибки при вызове readline, флаг O_NONBLOCK для дескриптора readfifo следует снять, перед тем как вызывать эту функцию.

6. Если клиент откроет свой канал на чтение перед открытием канала сервера, все зависнет. Единственный способ избежать блокировки заключается в вызове open для этих двух каналов в порядке, показанном в листинге 4.11, или в использовании флага отключения блокировки.

7. Исчезновение пишущего процесса воспринимаетсячитывающим как конец файла.

8. В листинге Г.3 приведен текст соответствующей программы.

9. Вызов select возвращает информацию о возможности записи в дескриптор, но вызов write приводит к отправке сигнала SIGPIPE. Это описано в книге [24, с. 153-155]; когда возникает ошибка чтения или записи, select возвращает информацию о том, что дескриптор доступен, а собственно ошибка возвращается уже вызовами read или write. В листинге Г.4 приведен текст соответствующей программы.

Глава 5

1. Сначала создайте очередь, не указывая никаких атрибутов, а затем вызовите mq_getattr для получения атрибутов по умолчанию. Затем удалите очередь и создайте ее снова, используя значения по умолчанию для всех неуказанных атрибутов.
2. Для второго сообщения сигнал не отправляется, поскольку регистрация снимается после отправки первого сигнала.
3. Для второго сообщения сигнал не отправляется, поскольку в момент отправки этого сообщения очередь не была пуста.
4. Компилятор GNU C в системе Solaris 2.6 (в котором обе константы определены как вызовы sysconf) возвращает ошибки:
5. В Solaris 2.6 мы указываем 1000000 сообщений по 10 байт в каждом. При этом создается файл размером 20000536 байт, что соответствует результатам, полученным с помощью программы 5.4: 10 байт данных на сообщение, 8 байт дополнительной информации (возможно, указатели), еще 2 байта добавочной информации (возможно, дополнение до кратного 4) и 536 байт добавочных данных на весь файл. Перед вызовом mq_open размер программы, выводимый ps, равнялся 1052 Кбайт, а после создания очереди размер вырос до 20 Мбайт. Это заставляет предположить, что очереди сообщений Posix реализованы через отображение файлов в память, и mq_open отображает файл в адресное пространство вызвавшего процесса. Аналогичные результаты получаются в Digital Unix 4.0B.
6. Размер аргумента, равный нулю, не вызывает проблем с функциями ANSI C memXXX. В оригинале стандарта 1989 года X3.159-1989 (ISO/IEC 9899:1990) ничего не говорилось по этому поводу (как и в документации), но в Technical Corrigendum Number 1 явно говорится, что указание размера 0 не вызовет проблем (но аргументы и указатели при этом должны быть правильными). Вообще, за информацией по языку C лучше всего обращаться по адресу <http://www.lysator.liu.se/c/>.
7. Для двустороннего взаимодействия процессов требуется наличие двух очередей сообщений (см. например, листинг A.15). Если бы мы изменили листинг 4.4 для использования очередей сообщений Posix вместо каналов, мы бы увидели, что родительский процесс считывает то, что сам же и отправил.
8. Взаимное исключение и условная переменная помещаются в отображаемый файл, совместно используемый всеми процессами, открывающими очередь. Очередь может быть открыта и другими процессами, поэтому при закрытии дескриптора очереди взаимное исключение и условная переменная не уничтожаются.
9. Массиву нельзя присвоить значение другого массива с помощью знака равенства в языке C, а вот структуре можно.
10. Функция main проводит большую часть времени заблокированной в вызове select, ожидая возможности чтения из конца канала. Каждый раз при получении сигнала возврат из обработчика прерывает вызов select, что приводит к возврату ошибки EINTR. Чтобы избежать этой проблемы, функция-обертка Select проверяет возврат данного кода ошибки и снова вызывает select, как показано в листинге Г.5. В книге [24, с. 124] вы можете найти более подробный рассказ о прерывании системных вызовов.

Глава 6

1. Оставшиеся программы должны принимать идентификатор очереди в числовом виде (вместо полного имени). Это изменение можно осуществить путем добавления нового аргумента командной строки или с помощью предположения, что полное имя, состоящее из одних цифр, является не именем файла, а идентификатором очереди. Поскольку большинство имен файлов, передаваемых ftok, являются абсолютными, они заведомо содержат по крайней мере один нецифровой символ (слэш), и это предположение является вполне корректным.
2. Передача сообщений с типом 0 запрещена, а клиент никогда не может иметь идентификатор 1, поскольку этот идентификатор обычно принадлежит процессу init.
3. При использовании единственной очереди на рис. 6.2 злоумышленник мог повлиять на все прочие процессы-клиенты. Если у каждого клиента есть своя очередь (рис. 6.3), злоумышленник портит только свою.

Глава 7

2. Вероятно, процесс завершит работу, прежде чем потребитель успеет сделать все, что нужно, поскольку вызов exit завершает все выполняющиеся потоки.
3. В Solaris 2.6 удаление вызова функций типа destroy приводит к утечке памяти, из чего становится ясно, что функции init осуществляют динамическое выделение памяти. В Digital Unix 4.0B такого не наблюдается, что указывает на разницу в реализации. Тем не менее вызывать функции destroy все равно нужно. С точки зрения реализации в Digital Unix 4.0B используется переменная типа attr_t как объект, содержащий атрибуты, а в Solaris 2.6 эта переменная представляет собой указатель на динамически создаваемый объект.

Глава 9

1. В зависимости от системы может потребоваться увеличивать счетчик до значений, больших 20, чтобы наблюдать ошибку.

2. Для отключения буферизации стандартного потока мы добавляем строку

к функции main перед циклом for. Это не должно ни на что влиять, поскольку вызов printf только один и строка завершается символом перевода. Обычно стандартный поток вывода буферизуется построчно, поэтому в любом случае один вызов printf превращается в один системный вызов write.

3. Заменим printf на

и объявим с как целое, а ptr — как char*. Если мы вызвали setvbuf для отключения буферизации стандартного потока вывода, библиотека будет делать системный вызов для вывода каждого символа. На это требуется больше времени, что дает ядру больше возможностей на переключение контекста между процессами. Такая программа должна давать больше ошибок.

4. Поскольку несколько процессов могут заблокировать на чтение одну и ту же область файла, в нашем примере это эквивалентно полному отсутствию блокировок.

5. Ничего не изменится, поскольку флаг отключения блокировки для дескриптора никак не влияет на работу рекомендательной блокировки fcntl. Блокирование процесса при вызове fcntl определяется типом команды: F_SETLKW (которая блокируется всегда) или F_SETLK (которая не блокируется никогда).

6. Программа loopfcntlnonb работает как положено, поскольку, как мы показали в предыдущем примере, флаг отключения блокировки никак не влияет на блокировку fcntl. Однако этот флаг влияет на работу loopnononenonb, которая не пользуется блокировкой. Как говорилось в разделе 9.5, неблокируемый вызов write или read для файла с включенной обязательной блокировкой приводит к возврату ошибки EAGAIN. Мы увидим одно из следующих сообщений:

и мы можем проверить, что это сообщение соответствует EAGAIN, выполнив

7. В Solaris 2.6 обязательная блокировка увеличивает время работы на 16%, а время процессора — на 20%. Пользовательское время процессора остается прежним, поскольку проверка осуществляется в ядре, а не в процессе.

8. Блокировки выдаются процессам, а не потокам.

9. Если бы работала другая копия демона, а мы открыли бы файл с флагом O_TRUNC, это привело бы к удалению идентификатора процесса из файла. Мы не имеем права укорачивать файл, пока не убедимся, что данная копия является единственной.

10. Лучше использовать SEEK_SET. Проблема с SEEK_CUR заключается в том, что этот вариант зависит от текущего положения в файле, устанавливаемого 1 seek. Однако если мы вызываем 1 seek, а потом fcntl, мы делаем одну операцию в два вызова и существует вероятность, что другой процесс в промежутке между вызовами изменит текущий сдвиг вызовом lseek. Вспомните, что все потоки используют общие дескрипторы. Аналогично, если указать SEEK_END, другой процесс может дописать данные к файлу, прежде чем мы получим блокировку, и тогда она уже не будет распространяться на весь файл.

Глава 10

1. Вот результат работы в Solaris 2.6:
2. Это не вызывает проблем с учетом правил, которые были указаны при описании `sem_open`: если семафор уже существует, он не инициализируется. Поэтому только первая из четырех программ, вызывающих `sem_open`, инициализирует семафор.
3. Это проблема. Семафор автоматически закрывается при завершении процесса, но значение его не изменяется. Это не дает другим программам получить блокировку, и все зависает.
4. Если мы не инициализируем дескрипторы значением -1, их значение оказывается неизвестным, поскольку `malloc` не инициализирует выделяемую память. Поэтому если один из вызовов `open` возвращает ошибку, вызовы `close` под меткой `errgog` могут закрыть какой-нибудь используемый процессом дескриптор. Инициализируя дескрипторы значением -1, мы можем быть уверены, что вызовы `close` не дадут результата (помимо возвращения игнорируемой ошибки), если дескриптор еще не был открыт.
5. Существует вероятность, что `close` будет вызвана для нормального дескриптора и вернет ошибку, изменив значение `errno`. Поэтому нам нужно сохранить это значение в другой переменной, чтобы оно не изменилось из-за побочного эффекта.
6. В этой функции ситуация гонок не возникает, поскольку `mkfifo` возвращает ошибку, если канал уже существует. Если два процесса вызывают эту функцию одновременно, канал FIFO создается только один раз. Второй вызов `mkfifo` приведет к возврату `EEXIST`.
7. В программе из листинга 10.22 ситуация гонок, описанная в связи с листингом 10.28, не возникает, поскольку инициализация семафора осуществляется записью данных в канал. Если процесс, создавший канал, приостанавливается ядром после создания, но перед записью данных, второй процесс откроет этот канал и заблокируется в вызове `sem_wait`, поскольку только что созданный канал будет пуст (пока первый процесс не поместит в него данные).
8. В листинге Г.6 приведена тестовая программа. Реализации Solaris 2.6 и Digital Unix 4.0B обнаруживают прерывание перехватываемым сигналом и возвращают ошибку `EINTR`.

Реализация с использованием FIFO возвращает `EINTR`, поскольку `sem_wait` блокируется в вызове `read`, который должен возвращать такую ошибку. Реализация с использованием отображения в память ошибки не возвращает, поскольку `sem_wait` блокируется в вызове `pthread_cond_wait`, а эта функция не возвращает такой ошибки. Реализация с использованием семафоров System V возвращает ошибку `EINTR`, поскольку `sem_wait` блокируется в вызове `semop`, которая возвращает эту ошибку.

9. Реализация с использованием каналов (листинг 10.25) является защищенной, поскольку таинственной является операция `write`. Реализация с отображением в память защищенной не является, поскольку функции `pthread_XXX` не являются защищенными и не могут вызываться из обработчика сигналов. Реализация с семафорами System V (листинг 10.41) также не является защищенной, поскольку `semop` не является защищенной функцией согласно Unix 98.

Глава 11

1. Нужно изменить только одну строку:
2. Вызов ftok вернет ошибку, что приведет к завершению работы обертки Ftok. Функция my_lock могла бы вызывать ftok перед semget, проверять, не возвращается ли ошибка ENOENT, а затем создавать файл, если он не существует.

Глава 12

1. Размер файла увеличится еще на 4096 байт (до 36864), но обращение к новому концу файла (36863) может привести к отправке сигнала SIGSEGV, поскольку размер области отображения в памяти равен 32768 байт. Причина, по которой мы говорим «может», а не «должен», — в неопределенности размера страницы памяти.

2. На рис. Г.1 показана схема с очередью сообщений System V, а на рис. Г.2 — с очередью сообщений Posix. Вызовы `memcp` в отправителе происходят внутри функций `mq_send` (листинг 5.26), а в получателе — внутри `mq_receive` (листинг 5.28).



Рис. Г.1. Отправка сообщений в очередь System V



Рис. Г.2. Отправка сообщений через очередь Posix, реализованную с mmap

3. Любой вызов `read` для `/dev/zero` возвращает запрошенное количество нулей. Данные, помещаемые в этот файл, попросту сбрасываются (аналогично `/dev/null`).

4. В результате в файле получится 4 байта — все нули (предполагается 32-разрядное целое).

5. В листинге Г.7 приведен текст нашей программы.

Глава 13

1. В листинге Г.8 приведен текст измененной версии листинга 12.6, а в листинге Г.9 — текст новой версии листинга 12.7. Обратите внимание, что в первой программе мы устанавливаем размер объекта вызовом ftruncate; lseek и write использовать для этого нельзя.
2. Одна из возможных проблем при использовании `*ptr++` заключается в том, что указатель, возвращенный `mmap`, изменяется, что может помешать впоследствии вызвать `munmap`. Если этот указатель еще понадобится, лучше его сохранить или вовсе не изменять.

Глава 14

1. Нужно изменить только одну строку:

Глава 15

1. Аргументы будут иметь размер `data_size + (desc_numxsizeof(door_desc_t))` байт.
2. Вызывать `fstat` не нужно. Если дескриптор не указывает на дверь, вызов `door_info` возвращает ошибку `EBADF`:
3. Документация содержит неверные сведения. Стандарт Posix утверждает, что функция `sleep` приведет к приостановке вызвавшего потока.
4. Результат непредсказуем (хотя, скорее всего, будет выполнен дамп памяти), поскольку адрес процедуры сервера, связанной с дверью, в новом процессе будет указывать на совершенно случайную область памяти.
5. При завершении `door_call` из-за перехвата сигнала сервер следует уведомить об этом, поскольку поток, работающий с этим клиентом, получит запрос на отмену выполнения. В связи с листингом 15.18 мы говорили, что отмена для создаваемых библиотекой потоков по умолчанию отключена, следовательно, поток завершен не будет. Вместо этого происходит досрочный возврат из вызова `sleep(6)` в процедуру сервера, но она продолжает выполняться.
6. Вот что мы увидим:

Запустив сервер 20 раз подряд, мы получим 5 сообщений об ошибке. Предсказать такую ошибку заранее нельзя.

7. Нет. Все, что нужно, — включать возможность отмены каждый раз при вызове процедуры сервера, как мы делали в листинге 15.26. Хотя в этом случае и приходится вызывать `pthread_setcancelstate` каждый раз при запуске процедуры сервера, накладные расходы, скорее всего, будут невелики.
8. Чтобы проверить это, изменим код одного из серверов (скажем, из листинга 15.6) так, чтобы он вызывал `door_revoke` из процедуры сервера. Поскольку аргументом `door_revoke` является дескриптор двери, его придется сделать глобальным. Вот что получится при запуске клиента (из листинга 15.1) два раза подряд:

Первый вызов завершается успешно, что подтверждает наше предположение насчет `door_revoke`. Второй вызов возвращает ошибку `EBADF`.

9. Чтобы не делать дескриптор `fd` глобальным, мы воспользуемся указателем `cookie`, который можем передать `door_create` и который затем будет передаваться процедуре сервера при каждом вызове. В листинге Г.10 приведен текст сервера.

Мы легко могли бы изменить листинги 5.17 и 5.18, поскольку указатель `cookie` доступен функции `my_thread` (через структуру `door_info_t`), которая передает указатель на эту структуру создаваемому потоку (которому нужно знать дескриптор для вызова `door_bind`).

10. В этом примере атрибуты потока не меняются, поэтому их достаточно инициализировать лишь единожды (в функции `main`).

Глава 16

1. Программа отображения портов (port mapper) не проверяет серверы на работоспособность во время регистрации. После завершения сервера отображения остаются в силе, в чем мы можем убедиться с помощью программы grcinfo. Поэтому клиент, связывающийся с программой отображения портов, получит информацию, которая была актуальной до завершения сервера. Когда клиент попытается связаться с сервером по TCP, библиотека RPC получит RST в ответ на пакет SYN (предполагается, что другие процессы не успели подключиться к порту завершенного сервера), что приведет к возврату ошибки функцией clnt_create. Вызов по протоколу UDP будет успешен (поскольку устанавливать соединение не нужно), но при отправке дейтаграмм через устаревший порт ответ получен не будет и функция клиента выйдет по тайм-ауту.

2. Библиотека RPC возвращает первый ответ сервера клиенту сразу по получении, то есть через 20 секунд после вызова клиента. Следующий ответ будет храниться в сетевом буфере для данной конечной точки до тех пор, пока эта точка не будет закрыта или не будет выполнена операция чтения из нее библиотекой RPC. Предположим, что клиент отправит второй вызов серверу сразу после получения первого ответа. Если потеря в сети не произойдет, следующей прибывшей дейтаграммой будет ответ сервера на повторно переданную клиентом дейтаграмму. Но библиотека RPC проигнорирует этот ответ, поскольку XID будет совпадать с первым вызовом процедуры, который не может быть равным XID для второго вызова.

3. Соответствующая структура в C — это char c[10], но она будет закодирована XDR как десять 4-байтовых целых. Если вы хотите использовать строку фиксированной длины, используйте скрытый тип данных фиксированной длины.

4. Вызов `xdr_data` возвращает FALSE, поскольку вызов `xdr_string` (прочтайте содержимое файла `data_xdr.c`) возвращает FALSE.

Максимальная длина строки указывается в качестве последнего аргумента `xdr_string`. Если максимальная длина не указана, этот аргумент принимает значение 0 в дополнительном коде ($2^{32}-1$ для 32-разрядного целого).

5. Подпрограммы XDR проверяют наличие достаточного объема свободной памяти в буфере для кодирования данных и возвращают ошибку FALSE при переполнении буфера. К сожалению, отличить одну ошибку от другой для подпрограмм XDR невозможно.

6. В принципе, можно сказать, что использование последовательных номеров в TCP для обнаружения повторов эквивалентно кэшу повторных запросов, поскольку эти последовательные номера позволяют обнаружить любой устаревший сегмент. Для конкретного соединения (IP-адреса и порта клиента) размер этого кэша соответствует половине 32-разрядного последовательного номера TCP, то есть 2^{31} или 2 Гбайт.

7. Поскольку все пять значений для конкретного запроса должны в точности равняться пятью значениям в кэше, первое сравнение должно выполняться для того поля, которое может отличаться с наибольшей вероятностью. Реальный порядок сравнений в пакете TI-RPC таков: (1) XID, (2) номер процедуры, (3) номер версии, (4) номер программы, (5) адрес клиента. Разумно сравнивать XID в первую очередь, поскольку именно это значение меняется от запроса к запросу.

8. На рис. 16.5 имеется двенадцать 4-байтовых полей, начиная с поля флага и длины и включая 4 байта на аргумент типа long. Получается 48 байт. При использовании нулевой аутентификации данные о пользователе и проверочные данные будут отсутствовать. При этом они займут по 8 байтов: 4 байта на тип аутентификации (AUTH_NONE) и 4 байта на длину аутентификационных данных (0).

В переданном ответе (взгляните на рис. 16.7, но помните, что используется протокол TCP, поэтому 4 байта флага и длины будут идти перед XID) будет восемь 4-байтовых полей, начиная с поля флага и длины и заканчивая результатом типа long. Вместе они дают 32 байта.

При использовании UDP единственное отличие будет заключаться в отсутствии поля флага и длины (4 байта). При этом размер запроса будет 44 байта, а ответа — 28 байтов, что можно проверить с помощью `tcpdump`.

9. Да. Отличие в обработке аргументов у клиента и сервера не зависит от пакетов, передаваемых по сети. Функция `main` клиента вызывает функцию заглушки для отправки пакета, а функция `main` сервера вызывает функцию заглушки сервера для обработки этого пакета. Передаваемая по сети запись RPC определяется протоколом RPC, и ее содержимое остается неизменным вне зависимости от того, поддерживается ли многопоточность.

10. Библиотека XDR выделяет место под эти строки (динамически). Мы можем проверить это, добавив следующую строку к программе `read`:

Функция `sbrk` возвращает текущий адрес вершины сегмента данных программы, а функция `malloc` обычно выделяет память непосредственно под этим адресом. Запустив программу, получим:

Это показывает, что указатель `vstring_arg` указывает на область, выделенную `malloc`. Буфер `buff` размером 8192 байта занимает адреса с `0x25e48` по `0x27e47`, а строка помещается непосредственно под ним.

11. В листинге Г.11 приведен текст программы-клиента. Обратите внимание, что последним аргументом `clnt_call` является сама структура `timeval`, а не указатель на нее. Также отметьте, что третий и пятый аргументы `clnt_call` должны быть ненулевыми указателями на подпрограммы XDR, поэтому мы указываем в этих аргументах `xdr_void` (функция, которая ничего не делает). Вы можете проверить, что именно так нужно вызывать функцию без аргументов и возвращаемых значений, написав тривиальный файл спецификации RPC, определяющий такую функцию, запустив `grcsen` и посмотрев на содержимое созданной заглушки клиента.

12. Получающийся размер дейтаграммы UDP ($65536 + 20 + \text{дополнительные расходы RPC}$) превосходит 65535 — максимальный размер дейтаграммы в IPv4. В табл. А.2 отсутствуют значения для Sun RPC с использованием UDP для сообщений размером 16384 и 32768, поскольку старая реализация RPCSRC 4.0 ограничивала размер дейтаграммы UDP некоторым значением около 9000 байт.

Литература

Для книг, статей и других источников, имеющих электронные версии, указаны адреса сайтов. Они могут меняться, поэтому следите за списком обновлений на сайте автора книги <http://www.kohala.com/~rstevens>.

1. *Bach M.J.* The Design of the UNIX Operating System //Prentice Hall, Englewood Cliffs, N.J., 1986.
2. *Birrell A. D., Nelson B.J.* Implementing Remote Procedure Calls //ACM Transactions on Computer Systems, vol. 2, no. 1, pp. 39-59 (Feb.), 1984.
3. *Butenhof D. R.* Programming with POSIX Threads //Addison-Wesley, Reading, Mass, 1997.
4. *Corbin J. R.* The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls //Springer-Verlag, New-York, 1991.
5. *Garfinkel S. L, Spafford E. H.* Practical UNIX and Internet Security, Second Edition //O'Reilly & Associates, Sebastopol, Calif, 1996.
6. *GoodheartB.,Cox J.* The Magi Garden Explained: The Internals of UNIX System V Release 4, An Open Systems Design //Prentice Hall, Englewood Cliffs, N.J., 1994.
7. *Hamilton. G., Kougiouris P.* The Spring Nucleus: A Mikrokernell for Objects // Proceedings of the 1993 Summer USENIX Conference, pp. 147-159, Cincinnati Oh, 1993.
<http://www.kohala.com/~rstevens/papers.others/springnucleus.1993.ps>
8. IEEE 1996. Information Technology — Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) //IEEE Std 1003.1, 1996 Edition, Insitute of Electrical and Electonics Enibeers, Piscataway, N.J. (July).
Данная версия Posix.1 (называемая также ISO/IEC 9945-1: 1996) содержит базовый интерфейс API (1990), расширения реального времени 1003.1b (1993), программные потоки Pthreads 1003.1c (1995) и технические поправки 1003.1i (1995). Чтобы сделать заказ, обратитесь на сайт <http://www.ieee.org>. К сожалению, стандарты IEEE не распространяются свободно через Интернет.
9. *Josey A.* Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification //Prentice Hall, Upper Saddle River, N. J., ed. 1997.
По адресу <http://www.UNIX-systems.org/online.html> можно найти множество спецификаций Unix (например, все технические руководства).
10. *Kernighan B.W., Pike R.* The UNIX Programming Environment //Prentice Hall, Englewood Cliffs, N. J., 1984.11.
11. *KernighanB.W., Ritchie D. M.* The C Programming Language, Second Edition // Prentice Hall, Englewood Cliffs, N.J., 1988.
12. *Kleiman S., Shah D., Smaalders B.* Programming with Threads //Prentice Hall, Upper Saddle River, N. J., 1996.
13. *LewisB., Berg D.J.* Multithreaded Programming with Pthreads //Prentice Hall, Upper Saddle River, N. J., 1998.
14. *McKusick M. K., Bostic K., Karels M.J., Quaterman J. S.* The Design and Implementation of the 4.4BSD Operating System //Addison-Wesley, Reading, Mass, 1996.
15. *McVoy L, Staelin C.* lmbench: Portable Tools for Performance Analysis //Proceedings of the 1996 Winter Technical Conference, pp. 279-294, San Diego, Calif, 1996.
Комплект средств для тестирования можно загрузить с сайта <http://www.bitmover.com/lmbench> вместе с книгой.
16. *Rochkind M.J.* Advanced UNIX Programming //Prentice Hall, Englewood Cliffs, N.J., 1985.
17. *Salus P. H.* A Quarter Century of Unix //Addison-Wesley, Reading, Mass, 1994.
18. *Srinivasan R.* RPC: Remote Procedure Call Protocol Specification Version 2 // RFC 1831, 18 pages (Aug.), 1995.
19. *Srinivasan R.* XDR: External Data Representation Standard //RFC 1832, 24 pages (Aug.), 1995.

20. *Srinivasan R.* Binding Protocols for ONC RPC Version 2 //RFC 1833, 14 pages (Aug.), 1995.
21. *Stevens W. R.* Advanced Programming in the UNIX Environment //Addison-Wesley, Reading, Mass, 1992.
22. *Stevens W. R.* TCP/IP Illustrated, Volume 1: The Protocols //Addison-Wesley, Reading, Mass, 1994.
23. *Stevens W. R.* TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols //Addison-Wesley, Reading, Mass, 1996.
24. *Stevens W. R.* UNIX Network Programming, Volume 1, Second Edition, Networking APIs: Sockets and XTI //Prentice Hall, Upper Saddle River, N.J., 1998.
25. *Vahalia U.* UNIX Internals: The New Frontiers //Prentice Hall, Upper Saddle River, N.J., 1996.
26. *White J. E.* A High-Level Framework for Network-Based Resource Sharing // RFC 707, 27 pages (Dec), 1975.
<http://www.kohala.com/~rstevens/papers.others/rfc707.txt>
27. *Wright G. R., Stevens W. R.* TCP/IP Illustrated, Volume 2: The Implementation // Addison-Wesley, Reading, Mass, 1995.

Примечания

Все исходные тексты, опубликованные в этой книге, вы можете найти по адресу
<http://www.piter.com/download>.

Проблема о порядке байтов в слове сродни проблеме лилипутов из «Путешествий Гулливера» Д. Свифта, которые никак не могли договориться, с какого конца начинать есть яйцо. Именно оттуда англоязычные программисты взяли термины little-endian (остроконечник) и big-endian (тупоконечник), подразумевая «little-end-first» и «big-end-first» (младший или старший байт идет первым). — *Примеч. перев.*